A reflective architecture to support dynamic software evolution

Rank, Stephen

A Reflective Architecture to Support Dynamic Software Evolution

Ph.D. Thesis

Stephen Rank,
Department of Computer Science,
University of Durham.

2002

Abstract

This thesis presents work which is concerned with the run-time evolution of component-based software systems. In particular, the main result of the research presented here is a framework which is used to model and control the architecture of a software system. This framework allows the run-time manipulation of the components which make up a software system. The framework makes the architecture of software systems visible, and allows interaction with it, using a reflective meta-object protocol.

The motivating objectives of this work are providing a framework to support *architectural flexibility*, *higher-level intervention*, *safe changes*, and *architectural visibility* in software systems.

The framework's behaviour and structure was motivated by a set of case-studies which have been used to guide its development and enhancement. The framework was developed iteratively, using each case-study in turn to evaluate its capabilities and to prompt the direction of development.

A detailed set of evaluation criteria are developed, and the framework is evaluated with respect to these. The framework was found to meet each of the four objectives fully, with the exception of the aim to allow only *safe changes* which is only partly satisfied. Ways in which the framework can be improved in order to more fully satisfy its objectives are suggested, as are other extensions to its behaviour.

Copyright

Declaration

No part of the material contained in this thesis has been submitted for a degree in any university.

The research documented here has, in part, been presented in the following publications:

- S. Rank, K. Bennett, and S. Glover. Flexx: Designing software for change through evolvable architectures. In *Systems Engineering for Business Process Change: Collected Papers from the EPSRC Research Programme*, pages 38–50. Springer-Verlag, 2000.

- K. Bennett, S. Glover, X. Li, and S. Rank. Designing software for change: Evolvable architectures. In *Proceedings of the ICSE 1999 Workshop on Software Evolution.*

Acknowledgements

This thesis could not have been produced without the large amount of support, help, and encouragement which I have received, and for which I am very grateful.

My supervisor, Professor Keith Bennett, is due grateful thanks for his constant encouragement, feedback, and support. Dr Simon Smith was very helpful in the very early stages of my work, and helped to provide structure and organisation for what was a disorganised collection of thoughts and ideas. Dr Liz Burd was a great help and encouragement in the final stages, giving hope that there was light at the end of the tunnel and happily performing the arduous task of proofreading. I am very grateful to my examiners, Dr Pearl Brereton and Dr Mark Harman, for their insightful comments on an earlier version of this thesis, both during the examination and afterwards.

The members of the SEBPC community have provided a forum for many useful and interesting discussions and exchanges of ideas. The meetings which this group organised have helped to shape my ideas and have provided a chance to see things from different perspectives.

This research has been funded and made possible by EPSRC.

In the course of my research, I have made use of a large amount of software provided by the Open Source community. Without this software, the work would have been a lot less fun.

The members of the department of Computer Science at Durham University have created an atmosphere in which research has been an enjoyable and well-supported activity. In particular, the members of Research Institute for Software Evolution have provided many opportunities for fruitful discussion, and have not been afraid to ask difficult questions!

The research community at Durham, both past and present, has been a place of mutual encouragement and support. In particular, thanks are due to Steven Glover, Nicolas Gold, Antony Hofton, James Ingham, Stephan Jamieson, Claire Knight, Phyo Kyaw, and Xiang Li who have all, in one way or another, made my time as a research student more interesting, productive, worthwhile, and fun.

My most grateful thanks go to my family, who have been a constant source of all kinds of support, help, and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis presents work done in the field of component-oriented software evolution. The work is concerned with the flexible and safe run-time evolution of component-oriented systems. In particular, it is software *systems* that are of interest here. Software is generally composed from parts which interact in many ways. In this thesis, the interactions and the parts which participate in these interactions are of equal importance.

This chapter introduces the material covered in the rest of the thesis.

1.1 Research Aims

Software is inherently difficult to change [Schneidewind, 1987]. In particular, changes which have not been (and in many cases could not be) anticipated at design time are especially difficult to make correctly and safely. In order to tackle these problems, the use of an architectural vocabulary is useful. In this thesis, software is considered as being composed of com-

ponents and connectors, capturing the behavioural and structural elements respectively [Shaw and Garlan, 1996].

The research presented here aims at tackling the following objectives:

The research presented here aims at creating a flexible architectural framework which allows the maintainer to intervene in a software system at a higher-level than at source code. The framework makes the architecture of a system visible at run-time and forbids some categories of unsafe modification.

1.2 Research Methods

The research was carried out using a set of case studies. A framework was developed, and evaluated with respect to developing systems within this framework using a given architectural style. The case studies motivated the development of a framework which allows run-time architectural evolution. This framework encapsulates an architectural view of a software system. This view is made explicit at run-time, in order to provide the maintainer with a complete high-level view of the structure of a system at run-time. A presentation and evaluation of the overall results of the research, including a better understanding of the problems, the framework, are also given.

1.3 Results

The main contribution of the research presented in this thesis is the development of a reflective framework which supports the run-time evolution of object-oriented software systems. This framework allows a maintainer

to make changes to a software system at the architectural level, and disallows some changes which are considered unsafe. The framework presents the maintainer with a run-time view of the architecture of a system in order to allow its evolution.

1.4 Summary

Chapter 2 introduces related work from the literature. In particular, work related to software evolution, software architecture, and software reflection is discussed.

In chapter 3, the objectives of the research (architectural flexibility, higher-level intervention, safe changes, and architectural visibility) are presented in detail.

In chapter 4, a potential means of achieving these objectives is presented. This solution is a reflective framework for enabling and managing run-time evolution of software systems. The framework is designed to have the following features:

- Allow the modifiability of a system at run-time.
- Have an explicit run-time representation of the architecture of a software system.
- Disallow some kinds of unsafe change.
- Present an accurate run-time view of the architectural of a system to the maintainer.

In chapter 5, the development of the framework, is described. The development of the framework was motivated both by the above required features and by the implementation of a set of case-studies using the framework. The various entities which make up the framework are described, as is the influence which the case studies have had on the development of the framework.

Chapter 6 presents the results of the research. The objectives described in chapter 3 are used to define a detailed set of evaluation criteria. These criteria are used to evaluate the framework.

Finally, chapter 7 summarises the work, and identifies possibilities for further work.

Chapter 2

Related Work

2.1 Introduction

This chapter introduces relevant work from the literature. Section 2.2 identifies problems in the area of evolution and maintenance of software systems, section 2.3 introduces the concept of software architecture, and identifies work in this field which is of relevance. Finally, section 2.4 introduces software reflection and relates work in this area to software evolution and architecture.

2.2 Software Evolution and Maintenance

This section identifies key problems in the evolution and maintenance of software systems, introduces the fundamental ideas behind the study of software evolution, and describes some techniques which have been developed to tackle these problems. Section 2.2.1 describes studies of software sys-

tems which have undergone changes, section 2.2.2 introduces models of the evolution of software which have been developed from these studies, and section 2.2.3 describes some methods of managing the evolution of software systems.

2.2.1 Studies of Software Evolution

There have been many studies of the evolution of real software systems. In this section, two such studies are examined, and used to determine some of the issues of software evolution. The first study described concentrated on factors external to the software in question, while the second study was more concerned with the properties of the software itself.

Lientz and Swanson [Lientz and Swanson, 1980] studied a large number of software projects in many organisations. Each organisation—which was in some way involved in data-processing—responded to a questionnaire. The study showed that software maintenance consumes approximately half the time of programmers and system administrators in the organisations which responded. In larger organisations, a larger proportion of time is spent on maintenance, though results did vary by type of industry. The study showed that in organisations where the maintenance activity is separated from development, a smaller proportion of effort is spent on maintenance. The study was carried out in the late 1970s, and the technology used by the organisations reflects this. For example, change logs are handled manually, and implementation is done using languages like COBOL and FORTRAN. The authors conclude that larger and older systems have greater maintenance

problems than smaller and newer systems, and that personnel issues such as programmer skill and staff turnover are of importance in the quality of system maintenance.

Lehman and Belady [Lehman and Belady, 1985a] made a detailed study of the development of an individual software system (IBM's OS/360). In contrast to the method used by Lientz and Swanson, Lehman and Belady studied the software product itself. They examined the size of the system at each release point, and showed that the size (both number of modules and lines of code) and complexity of a system grows with each successive release, unless specific effort is made to reduce these factors. During this work, Lehman and Belady developed the idea of software system types, using the terms S-type, P-type, and E-type to describe the three types [Lehman and Belady, 1985b]. S-type programs are the simplest kind, being those programs which are formally defined as a function between input and output, with no reliance on or interaction with their environment. P-type programs are those which solve real-world problems, and must use heuristics to arrive at approximate solutions. Examples include weather forecasting and chess playing, where the input the the software is well-defined and well-formed, but in order to arrive at a useful solution in a reasonable amount of time, approximations must be used. E-type software is the most complex and most interesting kind of software. An E-type program is situated in and interacts with its environment, leading to feedback between the software and the 'real world'. Total correctness of an E-type system cannot be shown in the abstract: as it interacts with its environment, it can be only be shown to be effective in a particular situation.

The results of these studies motivated Lehman to develop his laws of software evolution [Lehman, 1979, Lehman et al., 1997, Lehman, 1996]. These laws describe the behaviour of software systems over time [Lehman, 1996]:

Continuing Change An E-type program must either adapt or become obsolescent.

Increasing Complexity Unless an evolving program has work done specifically to reduce its complexity, it will become more complex as a result of the evolution.

Self-Regulation The evolution process is self-regulating, with statistically determinable trends and invariants.

Invariant Work-Rate The average effective global activity rate is constant over the life-time of the system.

Conservation of Familiarity The content of successive releases is statistically invariant.

Continuing Growth Functional content of a system must increase with each release in order to satisfy user demands.

Declining Quality Unless an E-type program is rigorously maintained and updated to its changing environment, it will be perceived as declining in quality.

Feedback System The evolution process for E-type programs is multi-loop and multi-level. Successful management of the process depends on recognising and accounting for this fact.

Two of the key problems of maintenance are understanding the program in order to determine where to make changes, and validating the changed version of a program—determining that the correct changes and no others have been made [Baxter and Pidgeon, 1997].

One important cause of the difficulty of maintenance is the complexity of software systems [Jackson, 1998]; understanding a system in its entirety is often necessary before even a simple change can be made and validated.

2.2.2 Models of Software Evolution

As described in the previous section, there have been several studies of the evolution of software systems. These and other studies have lead to models of software evolution which have been used to manage and control software evolution.

The main kinds of models identified are process models and product models. Process models identify the mechanism by which the evolution is carried out, and product models identify the characteristics of the software which are important with respect to evolution.

2.2.3 Approaches to Software Evolution

There are two complimentary approaches to handling software evolution. The first approach, related to reverse engineering, is to take a piece of software and work with it (which is often referred to as legacy-system evolution), while the second approach, related to forward engineering, is to attempt to design software which is easy to change.

Whether a software system has been designed for ease of modification or not, there are common tasks which must be performed. In order to change a software system, the software engineer performing the task must understand both the system and the changes to be made [Takang and Grub, 1996]. The engineer must also be able to verify that exactly the required changes to behaviour have been made.

Various techniques for handling software evolution have been described in the literature, including those by Takang and Grub [Takang and Grub, 1996] and Pigoski [Pigoski, 1996]. Takang and Grub describe several software life-cycle processes, and put each in the context of systems which evolve, while Pigoski takes a more evolution-centred approach, concentrating more on the processes which occur after initial delivery of a software system. Pigoski describes software evolution processes, metrics, and management issues.

While developing software which is easy to change is not entirely removed from changing so-called 'legacy' software, it is sufficiently different to merit separate treatment. Various techniques for creating software have been described. These range from product-oriented guidelines for developing understandable source code [McConnell, 1993, Kernighan and Pike, 1999] to processes with attempts at psychological grounding in program comprehension [Smith, 1999].

There have been several attempts to categorise methods for dynamically changing software (in other words, making changes without halting the execution of the program). These include simple techniques based on plugins (or dynamically loadable modules) and parameter alteration [Rubini, 1997], and more sophisticated approaches based on component replacement or adap-

tion [Bihari and Schwan, 1991, Segal and Frieder, 1989].

2.3 Software Architecture

When a real software system is being dealt with, the designer is faced with issues relating to the structure and organisation of the system, not just with computational issues. These are considered to be architectural issues [Garlan and Shaw, 1994]. This section considers the use of the term 'architecture' as applied to software, and introduces concepts which have been used in research into software architecture. In general, architectural issues deal with high-level, abstract view of software systems [Shaw et al., 1995], relating to the overall structure of the system at the highest level. It has been claimed that the definition of the architecture of a system is a key milestone in the life of a software system [Gacek et al., 1995].

Initial work in applying architectural concepts to software was inspired by Alexander's work on architecture and town planning [Alexander et al., 1977]. This work inspired research into both software architecture and design patterns for software engineering [Gamma et al., 1995, Beck and Johnson, 1994].

In this section, the use of architectural abstractions applied to software engineering is examined, as is the idea of design patterns.

2.3.1 Definitions of Architecture

The term 'architecture', as applied to software, can be defined as "The structure(s) of a system, comprising software components, the externally visible properties of these components and the relationships among them"

[Bass et al., 1998]. This definition highlights three important aspects of architecture:

- System Structure
- Components
- Relationships

The *structure* of a system shows the *components* which make up the system and the *relationships* between them.

For any system, many views of the architecture can be identified and used to reason about the system [Perry and Wolf, 1992]. Example views include [Bass et al., 1998];

- Modular structure of the code.
- Conceptual structure of the system.
- Run-time process structure.
- Data-flow relationships.
- Control-flow within the system.
- 'Uses' relationships.

Each of these structures interacts, at design-time and at run-time.

2.3.2 Architectural Concepts

In this section, the key concepts used by software designers to describe the architecture of systems, and to talk about architectural issues in general, are discussed.

A key part of the architectural description of a software system is to break it down into two types of entity: components and connectors.

A software component (in the architectural sense) is a single unit of independent deployment [Szyperski, 1997]. Each component has a well-defined interface through which it interacts with connectors.

Software connectors explicitly represent and mediate the communication that occurs between components in a software system [Oreizy et al., 1998b], and allow the separation of behavioural and interfacing requirements of components [Oreizy and Medvidovic, 1998]. As with components, connectors can be composed [Garlan, 1998].

2.3.2.1 Architectural Styles and Patterns

An architectural style is defined by the types of component that can take part in an architecture of that style, the topology of the style, a set of constraints on the interactions in the system and the types of connector by which these components can interact [Bass et al., 1998].

An architectural style defines a family of software systems which each have the same pattern of structural organisation [Garlan and Shaw, 1994].

Various architectural styles can be identified [Garlan and Shaw, 1994], including, for example;

Pipe-and-Filter Data is passed along pipes and processed by filters.

Implicit Invocation In this style, event-based, (multi-cast) call-backs are used to broadcast events to each component in the system.

Layering Each layer in a system communicates only with those layers on either side of it.

2.3.3 Modelling Software Architecture

There are many methods in use for describing the architecture of a software system. These range from informal diagrams of boxes and lines (which Shaw and Garlan claim cannot properly be called architectural descriptions [Garlan and Shaw, 1994]) to formally defined architecture description languages such as Darwin [Magee et al., 1995].

2.3.3.1 Architectural Description Languages

Programming languages do not allow the description of software architecture. For this reason, various languages to describe software architecture have been created. These are variously known as architecture description languages, module interconnection languages, and configuration languages. These languages serve to capture the architecture of a software system, and to allow automatic construction of the system from the components. In addition, the formal (or semi-formal) nature of these languages provides scope for automatic verification of system properties [Prieto-Diaz and Neighbours, 1986], such as type-safety.

To be useful, an architecture description language must allow the following [Shaw and Garlan, 1994]:

- Composition of components and connectors.
- Abstraction to the design (rather than implementation) level.
- Reusability of design patterns and elements.
- Configuration of system structure.
- Heterogeneity; use of different patterns in the same design and different implementation languages.
- Analysis—both automated and manual—of architectural qualities, including dynamic properties of systems.
- Precision in system description

Darwin [Magee et al., 1995] has a well-defined syntax (both textual and diagrammatic) and semantics (described in terms of Milner's π-calculus). Darwin is based on a service-oriented view of software architecture: each component provides and requires services as output and input respectively. Components are described in terms of these inputs and outputs, while binding is described separately.

Rapide [Luckham et al., 1995] is a language designed to be used to prototype architectures. Architectures are described in terms of the components which are provided. There are several parts to the language, including a pattern language, interface-definition language, and an executable language

which allows the composition of components to provide compound components.

While Darwin is a declarative language aimed at modelling distributed and dynamic systems, Rapide is intended to provide an executable description language for large-scale systems.

2.3.4 Architecture and Evolution

The lack of explicit representation of communication in a software system causes problems with the evolution of the system [Oreizy et al., 1998b]. Maintaining the existence of connectors through to the run-time instantiation of the code allows connectors to encapsulate more information about the communication that occurs between components, to contribute to the mobility, distribution and extensibility of systems, and to act as domain translators (providing mappings from messages in one format to messages in another) [Oreizy et al., 1998b].

The initial design of a modern system usually aims to have low inter-component coupling. This coupling between modules increases as a system is maintained [Lehman, 1998b].

Whatever the initial architecture of a software system, maintenance of the system without regard to the effects on the architecture will cause degradation of architecture [Lehman, 1996]. There are several ways to tackle the problems here:

- Use a process of maintenance that pays explicit and careful attention to the architecture of the system.

- Design the architecture of the system in such a way that maintenance can be carried out in a way that preserves the structure and 'cleanliness' of the architecture.

When building a software system of significant size, reuse of existing pieces of software is desirable. Usually, unless the components have been specifically designed to work together and do not violate each others' assumptions, simple composition of components is not possible. Each component will make different assumptions about the environment and the behaviour of other components in the system, leading to so-called architectural mismatch [Garlan et al., 1995]. The most common approach to tackling this mismatch is to 'wrap' components (commonly by inserting 'glue' code between them) to insulate them from each other and to transform the input and output [Shaw, 1995].

One approach to architectural reuse is the concept of product-line architectures. These provide the opportunity to reuse parts of previously existing systems in later software, though this requires a significant amount of work to achieve, and is hard to perform after-the-fact [Bosch, 1999].

2.3.5 Models of Architectural Evolution

Use of the C2 architectural style [Oreizy et al., 1998b], which is based on a layered system of components and connectors, has been claimed to ease *run-time* software evolution; evolution without re-compilation of the system, in such a way that the system retains its integrity without becoming successively brittle over modifications [Oreizy and Medvidovic, 1998]. Two types

of system change are identified: changes to the system requirements, and changes to the implementation that do not affect the requirements.

Work on run-time architectural evolution has, in general, concentrated on providing the ability to dynamically replace components. This typically requires provision to be made at design-time [Amador et al., 1991, Oreizy, 1998].

Distributed systems offer further challenges and opportunities. Large distributed (and other) systems may need to remain functional for long periods of time without interruption. In order to tackle this, Kramer and Magee propose replacing traditional (build-time) static configuration with incremental dynamic (re-)configuration [Kramer and Magee, 1985]. This requires a greater separation between programming (implementation of behaviour) and configuration (implementation of composition), and requires a configuration language distinct from the programming language(s) used in the system. The C2 architectural style provides explicit representation of connectors, which provides the ability to abstract away from distribution and to insulate components from changes occurring in other parts of the system [Oreizy and Taylor, 1998a, Oreizy and Taylor, 1998b].

2.3.6 Architecture and Components

Traditional programming languages have little (if any) support for architectural (rather than modular) composition of software. Component-oriented software development can help to address this. The ideas of giving components interfaces (using an interface definition language) is also useful. Current component models (such as CORBA and COM) do not provide architectural

concepts [Oreizy et al., 1998a].

2.4 Reflection

A reflective computational system is able to examine and adapt its own state and behaviour [Sobel and Friedman, 1996]. Reflective capabilities have been added to many programming languages, especially Lisp [Kiczales et al., 1991] and Smalltalk [Goldberg and Robson, 1983]. In this section, work on reflection is examined, with particular emphasis on work related to software architecture and evolution.

In a reflective software system, there are two distinct kinds of entity, which are thought of as belonging to two separate layers [Cazzola et al., 1998, Steindl, 1997]:

Base-level entities are those which provide the computational components of the software system.

Meta-level entities operate on the base-level entities, treating them as data.

There are many reasons to use reflection. An example of its use includes tailoring the implementation of a programming language for efficiency [Kiczales et al., 1993]. In a reflective tower, each meta-level entity can be considered as a base-level entity with respect to a higher-level interpreter. In other words, reflection is based on the observation that what is considered a program by the programmer is treated as a data item by the language tools (*e.g.*, interpreter, compiler).

Reflection makes explicit properties of and structures within software that previously have been implicit[Kiczales et al., 1991, Cazzola et al., 1998]. Reflection also allows encapsulation of aspects of software which are subject to change [Buschmann, 1996].

2.4.1 Types of Reflection

There are various ways of dividing the field of reflection. The two main axes are:

Structural *vs* behavioural reflection [Kirby et al., 1998] In a system which exhibit structural reflection, the meta-object(s) hold information about the organisational structure of the base-level components. In a behaviourally reflective system, the behaviour of the base-level objects is represented at the meta-level. These two types of reflection can be combined; in one sense, behavioural reflection is concerned with lower-level properties than structural reflection.

Reflective tower *vs* meta-circular interpreter [Smith, 1982] Using a reflective tower, a meta-layer is distinct from its base layer. Each layer can have a meta-level object, leading to a (conceptually unbounded) tower of reflection. If the meta-circular (or introspective) approach is used, the base layer and the reflective layer are the same thing; entities can operate on themselves.

2.4.1.1 Structural and Behavioural Reflection

Structural reflection has been included in an extension of the Java programming language [Golm and Kleinöder, 1998] known as 'metaXa'. This is in addition to the reflective capabilities already present in standard Java, and allows more than one meta-object per object. Behavioural reflection can also be added to Java [Welch and Stroud, 2001], allowing the programmer to alter the behaviour of the virtual machine at run-time. This approach can be used to implement a security mechanism for mobile code [Welch and Stroud, 2000].

2.4.1.2 Towers and Meta-Circular Reflection

Tower reflection (also known as meta-circular reflection) is the more common kind of reflection in use. Smith [Smith, 1982] introduced reflection into Lisp as a means of allowing the programmer to modify the language. Kiczales [Kiczales et al., 1991, Kiczales et al., 1993], likewise, used a meta-object protocol to give the programmer access to the implementation of the language in which they are programming. In a reflective tower, each program is considered to be implemented in an 'interpreter', which is also a program; each interpreter is also interpreted, leading to a conceptually infinite 'tower' of interpreters [Mendhekar and Friedman, 1993, Danvy and Malmkjær, 1988].

2.4.2 Applications of Reflection

Several languages have reflective capabilities built into them, including well-known languages such as Smalltalk, Java, Lisp, Oberon, and research languages such as Beta, metaXa, and Kava. Due to the need for some degree

of self-reference, most languages which allow reflection are (to some degree) interpreted.

In Smalltalk, a simple meta-object protocol is used to allow structural reflection on classes [Goldberg and Robson, 1983]. Theses features are generally only used by the language interpreter, though they are exposed for use by any program.

The Java language allows structural reflection on objects and classes, and this can be extended (to allow for greater control of the security features of the virtual machine [Welch and Stroud, 2001, Welch and Stroud, 2000], for example).

In Beta [Brandt, 1995, Brandt and Schmidt, 1995], reflection is used to generalise the type system and to allow experimentation with the implementation of the language.

In Oberon [Steindl, 1997] (a strongly-typed language, in contrast to, for example, Lisp), reflection has been used to experiment with type-safe meta-object access across module boundaries.

2.4.3 Reflection and Evolution

The use of reflection in a software system has the following consequences for evolution [Buschmann, 1996]:

- Modification can occur at a higher level than the source. Changes are made at the meta-object level, which can enforce constraints on the type of changes which can be made.

- Modification is less complex. Changes are made at a level which cor-

responds more closely to the level at which changes are specified.

- Changes are more constrained and thus safer. The classes of changes which can be made are limited to those which are catered for at design-time. Encapsulation and abstraction are preserved.

- Some changes which were unforeseen at design time can be easily made. Although these changes are limited to the class of changes made available, some changes which have not been (explicitly or implicitly) foreseen at design- and implementation-time can be made.

One use of a reflective meta-object protocol is to open up aspects of the programming language in which a system is written [Kiczales et al., 1993]. This enables evolution of the system at run-time. Typically, every object in a reflective system has an associated meta-object. Each of these meta-objects contains data about the class of the object, its public interface (return and parameter types of methods), and other information. Often (as is the case in Java), classes have meta-objects, which contain information which is true for every object of that class. Use of this information aids in ensuring consistency of a system (making sure that interfaces match, and so on) when changes are made, particularly run-time changes.

Using reflection, the implementation of a system can be opened up in a more controlled way than so-called 'glass box' reuse, avoiding some of the restrictions of 'black-box' reuse [Kiczales, 1996]. This allows some form of adaptation of implementation as well as interface [Maeda et al., 1997]. Boyapati has proposed that the Java Virtual Machine be modified to allow

this form of introspection [Boyapati, 2002], enabling parameterised polymorphism and persistence, for example, to be added to Java.

2.4.4 Reflection and Architecture

Structural reflection has been used to allow adaptation of software architecture [Welch and Stroud, 1998]. This work allows the run-time adaptation of connectors in a software system. In addition, it has also been suggested that a meta-object protocol can be used to allow the composition of components by allowing their modification [Heineman, 1998, Sabry, 1998, Mätzel and Bischofberger, 1996].

Cazzola *et al* describe a system in which a software system maintains an explicit architectural model of itself, allowing inspection of this model at run-time [Cazzola et al., 1998].

2.5 Summary

This chapter has introduced relevant work from the literature, with particular emphasis on work related to software evolution, software architecture, and software reflection. This material forms the background, motivation, and basic material for the work presented in the rest of this thesis.

Chapter 3

Problems in Software Evolution and Software Architectures

3.1 Introduction

This chapter presents the following problems, which will be addressed in later chapters.

This chapter details the objectives of the research presented in this thesis. These objectives are known as *architectural flexibility*, *higher-level intervention*, *safe changes*, and *architectural visibility.*

These problems are of importance to software evolution, which is a major cost factor for the software industry [Lientz and Swanson, 1980]. In order to tackle these problems, an architectural view of software is taken, considering the structure and organisation of software systems. The integrity of the structure of software is at least as important—in terms of the evolutionary characteristics of such a system—as its functional behaviour [Brooks, 1995,

Dijkstra, 1968], particularly for large-scale software engineering projects.

The research areas that are presented here are associated with software evolution; that is, the process of modifying software artefacts. The artefacts, rather than the processes, of software engineering are considered; of interest here is the result, not the means by which the results are generated. This point is expanded in section 3.2.4.3. Software engineering has traditionally been concerned with the creation of new software artefacts, rather than adapting current artefacts to new purposes; failure to pay sufficient attention to the fact that software must continually change after its installation [Lehman and Belady, 1985b] is costly. Most (typically 50–70%) of the effort and expense associated with a piece of software is spent after the initial installation into its environment [Lientz and Swanson, 1980]. For this reason, problems of software evolution are of importance.

This chapter identifies four desirable properties of software evolution, namely *architectural flexibility*, *higher-level intervention*, *safe changes*, and *architectural visibility*.

These properties will be addressed in the following chapters. These properties are necessary in order to reduce the cost of software maintenance.

3.2 Context and Assumptions

The work described in this thesis is in the field of software evolution. The use of software architectural concepts are of key importance to the research. A software system, is composed of two types of entity; computational entities (known as *components*), and communicational entities (known as *connec-*

tors). Taking a high-level, design-oriented view of a software system, these two categories of entity can be viewed as the sole constituents of software. Evolution has fundamental effects at the design level as well as at the code level. Because of this, it is essential that change to software systems is considered at the architectural level as well as at lower levels of abstraction.

The following assumptions are made in order to simplify the research and to situate it within the area of the software evolution field concerned with the evolution of software (in particular the architecture of software systems):

Architectural Abstraction It is possible to identify an architectural overview of a system.

Control The evolution of a system is carried out in a controlled manner, by suitably-qualified personnel.

Designed Behaviour A system is designed in order to satisfy a given set of requirements, and does not adapt itself over time to fulfil differing requirements.

These assumptions are examined in the following sections.

3.2.1 Architectural Abstraction

When considering a software system (We consider a software system to be one which mainly consists of software components. Hardware components, *e.g.*, CPUs, are considered part of the infrastructure.), it is possible to identify an abstraction which can be referred to as the 'architecture' of that system. A more rigorous definition of the term 'architecture' is given in section 3.3.1.1;

the architecture of a system is a high-level design view showing the computational elements which make up the system, and the interactions between these elements [Magee et al., 1995]. This assumption allows the identification of a high-level abstract view of the overall structure of a software system. Identifying this structure is possible, in most cases, for new (*i.e.*, unmodified) systems. In some cases, decompositions smaller than the whole system but larger than individual lines of code are not possible; these systems are either pathological cases of bad design, or very small systems with no interesting decomposition [Burd and Munro, 1998].

3.2.2 Control

Evolution is carried out in a sound and well-thought-out fashion. Expert 'software architects', or senior designers, who have good and thorough knowledge of the system's architecture, plan and manage the changes. This can be referred to as the 'evolution' stage of the software lifecycle, when the key personnel who are associated with a software project are still available [Bennett and Rajlich, 2000]. This assumption does not apply after the structure of a system has degraded, as will happen if attention is not paid to maintaining this structure [Lehman, 1996].

3.2.3 Designed Behaviour

The systems of interest are designed to fulfil a given set of requirements; emergent behaviour of software systems (for example, software agent systems) is beyond the scope of the current research. In other words, the type of sys-

tem that is under consideration is not an adaptive or otherwise intelligent system; their behaviour does not change over time. This is the assumption that a particular software system has been designed with overall predictable behaviour in mind, not that individual components are not adaptive (for example, a system may have a speech-recognition component which uses an adaptive algorithm).

3.2.4 Software Engineering

In this section, problems related to software engineering (with respect to both process and product) are identified, with particular focus on software evolution.

3.2.4.1 Software Processes

The *process* of software engineering has been much documented. Most prescriptive approaches make little mention of evolution in their model; they concentrate instead on the forward development of software systems. In order to consider evolution as a primary property of the software process, it is possible to concentrate on evolutionary aspects of the life-cycle. Initial development can be thought of as a short-lived activity in a much longer software life-cycle in which most activity is concentrated on software evolution [Bennett and Rajlich, 2000].

There is some debate over the moment at which initial development ceases and a software products enters the 'maintenance' phase of its life. The standard IEEE definition of the term 'maintenance' is as the set of activities that

take place after the delivery of a piece of software [I.E.E.E., 1994], though some feel that it is initial development that is anomalous and that all software engineering is software evolution [Schneidewind et al., 1999]. Software 'maintenance' or evolution, consists of many of the activities (such as requirements capture, programming, and so on) that are carried out during development [McDermid, 1991], and so can be considered as part of the development process (or at least not independent of it). The concept of 'delivery' of a piece of software is muddled by the use of component-oriented software [Szyperski, 1997], which allow piecemeal 'delivery' of a software system, and using external components for some (not necessarily all) of the functionality of a software system.

3.2.4.2 Software Artefacts

The main kind of systems that are of concern here are those that are situated in an environment and interact with a non-empty set of users. This type of system is often referred to as an 'E-type' system [Lehman and Belady, 1985a], as distinct from 'S-type' software (which is software that can be formally defined by a mathematical specification) and 'P-type' software, which address problems that can be clearly defined (such as playing chess), but are only approximate solutions. E-type software is embedded into its environment, and embodies a view of that environment. As the software forms part of this environment, there is a feedback loop. A piece of software is finite, while its environment is potentially infinite. To bridge this gap, assumptions about the environment are made [Lehman, 1998a]. These assumptions cause problems during evolution when they become invalid [Lehman, 1989].

Any software system can be said to have an architecture. What is of interest here are properties of the system and its architecture. There are many kinds of architecture (as shown in section 3.3.1.1). Many of the properties—such as dependencies, performance, reliability, and so on—of a system under maintenance are determined by the properties of that system's architecture [Lung et al., 1997, Shaw and Garlan, 1996]. Determining the structure of a software system is a key part of any process of software evolution [von Mayrhauser and Vans, 1995].

Properties of a system can partly be determined from the structure (architecture), and partly from the behaviour (components) of the system. The structure and the behaviour of a system cannot be completely separated—communication between components is a significant factor in the behaviour of a system—though, for some purposes such as determining dependencies, the structure of the system can be considered separately from the behaviour of the components which make it up. A system is built up from three classes of entity: primitive expressions in some (programming) language, composite elements (such as modules, classes, libraries), and abstraction mechanisms by which composite elements are named and manipulated as entities in their own right [Abelson et al., 1985]. In order to tackle the problems of software evolution, it is necessary to consider both the structure and the behaviour of software systems.

It is not possible to ignore the structure of a software system and to concentrate on the behaviour of the system as a whole. To consider the organisation of a software system in conjunction with its behaviour brings benefits in terms of both forward development and reuse [Shaw, 1995]. The

infrastructure that supports—rather than provides—-the functionality of a system can comprise up to 90% of an application's code [Shaw, 1995].

The software evolution artefact problems that will be considered during this thesis are as follows:

- Software architectures are not flexible enough to allow insertion, removal, and update of software artefacts without causing the infrastructure to degrade. The ability to treat a system as a collection of parts, and to operate on each of these parts individually, to some extent, is necessary here. If anti-regressive work (*i.e.*, preventative maintenance) is not done on an architecture specifically in order to maintain the structure, that structure will degrade [Lehman, 1996].

- Intervention in software always takes place at the code level, not at the component level. This is due, in part, to the inflexibility of software architectures mentioned above. Code does need to be changed, but the impact of a change should (in the evolution stage of the software life-cycle) be considered at the architectural level, not just on the level of code.

- Modification of software can have undesirable effects. For example, changes to a single component can cause 'ripple effects' throughout the rest of the system. In this context, the 'system' in consideration can include other parts of the organisations that interact with the software [Fyson and Boldyreff, 1998]. Reasoning about changes that 'ripple' outside the software is not tackled in the current research. The

ability to reason about these ripple effects is important, in order to minimise unwanted side-effects of operations.

- It is difficult to determine the architecture of a software system, particularly without supporting documentation.

There are many other problems that could be addressed, including the testing, distribution, and verification problems. These, and other, problems are beyond the scope of this thesis.

The rest of this chapter examines the problems identified above in more depth and identifies prerequisites for their solution and criteria for success.

3.2.4.3 Processes and Products

This research is concerned with the products of software engineering, rather than the processes by which they are constructed, used, and changed. The software process is a multi-level, multi-loop feedback process [Lehman, 1997]. Much data is transferred in both directions between users, developers, managers, *etc.*, relating to satisfaction of many goals, not all of which are compatible. This applies particularly when the whole life-cycle (including evolution) is considered [Lehman, 1997], and is beyond the scope of this thesis. What is of concern here is the behaviour of software during and after changes have been made. In this thesis, it is the feedback at the level of product iterations that is of interest, as modelled by the spiral model of software engineering [Boehm, 1988].

Although the process by which software artefacts are generated is of great importance, the main concern in this thesis is the properties of artefacts, not

the means by which these properties are arrived at.

3.2.5 Software Evolution

As a software product gets older, it becomes less useful to its users unless it is modified [Lehman, 1996]. Software is thus required to change as the requirements change, in a continuing, self-stabilising, feedback cycle [Lehman, 1997]. Software evolution has been demonstrated to obey the laws shown in section 2.2.1;

The initial design of a modern system usually aims to have low inter-component coupling. This coupling between modules generally increases as a system is maintained [Lehman, 1998b].

As a system ages and is maintained, it becomes increasingly brittle *i.e.*, resistant to and more likely to require increasing amounts of corrective maintenance under change. This is due to violations of the architecture and insensitivity to the architecture during evolution (leading to the architecture becoming obscured) [Perry and Wolf, 1992].

In the context of software engineering, many entities evolve. Software itself evolves in response to users' requirements changing and feedback, as does the process of software engineering, the environment in which a software artefact is situated, the documentation that describes the structure and the usage of an artefact, or the users of an artefact. In this work, the evolution of software artefacts is of concern. Principally, software artefacts are software components and systems, but can also include test harnesses and suites, and also documentation of design decisions, requirements, and so on. Evolution-

ary pressure usually comes from sources external to the software system in question; corporate policy changes (in systems that relate to an organisation's business); changes in technology (*e.g.*, the introduction of object-oriented technology); or realisation that the original design and/or implementation is flawed or inadequate [Lehman, 1998b]. Environmental changes can invalidate assumptions that are embedded—sometimes implicitly—in a program. Lack of documentation is a major cause of some of the problems that occur in software maintenance [Baxter and Pidgeon, 1997].

In principle, a given piece of software can be changed into any other. However, for a given specification, there are many more incorrect programs than there are correct programs. Given an unrestricted landscape of syntactically correct programs, most do not satisfy a given set of requirements. Furthermore, most of the programs will fail at run-time due to semantic errors. Part of the aim of this work is to restrict directions in which a piece of software can be evolved, in order to make it easier to identify beneficial changes, and more difficult to perform harmful modifications.

When any form of architectural change occurs, it is important that the mechanics of the change are considered separately from the semantic effects of the change, and that changes must be reasoned about in order to verify that their effects on the system are exactly those which are required [Oreizy and Medvidovic, 1998].

3.3 Objectives

This section defines the aims of the research which is presented in following chapters. Each objective is defined (in terms of the concepts identified above), examples are given, and its consequences stated.

The objectives are:

Architectural Flexibility Creating software systems which have architectures that are easy to change.

Higher-Level Intervention Allowing software changes to be made at a higher-level than the source.

Safe Changes Disallowing certain kinds of unsafe changes to a software system's architecture.

Architectural Visibility Making the structure of a software system visible (from the software itself, automatically) at run-time.

These objectives are discussed in the following sections.

3.3.1 Flexible Architectures

Flexible architectures are necessary for software evolution. In order for software changes to be made at the component level, which more closely matches the conceptual level of system designers, the architecture of a system must accommodate modification without requiring that the person performing the modification intervene with the source code of components other than those which directly require modifications to their behaviour. It is not enough that

an architecture is clean and well-defined; the architecture of a system must be designed in such a way that changes to the system can be made with effort appropriate to the size of the change. Component-oriented engineering techniques, and before that, object-oriented programming, attempt, among other things, to address modularisation issues. This alone is necessary but not sufficient for flexibility.

3.3.1.1 Definitions

Architecture There is no single, concrete, well-defined, and universally-accepted definition of software architecture. It is therefore necessary to define here what is meant by the term 'architecture' in this thesis. Architecture is accepted as an abstraction from a real system [Shaw et al., 1995], though there are many ways of abstracting from any given system. The most generally accepted definitions of the term 'software architecture' are that the architecture of a system is concerned with organisational, rather than computational, issues [Garlan and Shaw, 1994], that it is external properties of components that is important when studying software architecture [Bass et al., 1998]. It is also generally accepted that software architecture involves decomposition of a software system into a set of components that interact *via* a set of connectors. Architecture is distinct from design in that an architecture is a higher-level construct, encapsulating detailed design. Architectural views of a software system give broader overviews than detailed design views. Decisions of what belongs in an architectural model and what should be left out (but included in the detailed design) are mainly matters of judgement for software engineers.

For any system, more than one kind of architectural perspective can be identified and used to reason about the system [Perry and Wolf, 1992]. Examples include [Bass et al., 1998];

- Modular structure of the code.
- Conceptual structure of the system.
- Run-time process structure.
- Data-flow relationships.
- Control-flow within the system.
- 'Uses' relationships.

In this thesis, the abstraction that will be used is the component-oriented structure of the system, with first-class connectors [Shaw, 1993] as data-flow and synchronisation constructs. The use of explicit connectors is a very important part of this work, as, in many systems, the lack of explicit representation of communication in a software system causes problems with the evolution of the system [Oreizy et al., 1998b]. These connectors will be explicitly represented at run-time, as the run-time modification of a system depends upon this.

An architectural style is defined by the types of component that can take part in an architecture of that style, the topology of the style, a set of constraints on the interactions in the system and the types of connector by which these components can interact [Bass et al., 1998].

An architectural style defines a family of software systems which each have the same pattern of structural organisation [Garlan and Shaw, 1994].

Various architectural styles, which identify the kind of components and connectors which make up a system, can be identified. For example, the following styles are common [Garlan and Shaw, 1994];

Pipe-and-Filter A data-flow style. Components are 'filters', with input and output ports which consume and produce data respectively. Data is carried along asynchronous ordered streams.

Implicit Invocation In this style, event-based, (multi-cast) call-backs are used to broadcast events to each component in the system. Components register (with an event-handler) their interest in event classes. When events occur, the event handler notifies those components which have registered their interest. This style is used in graphical user-interfaces.

Layering Each layer in a system communicates only with those layers on either side of it. In general, lower-level layers (such as operating system routines) tend to be called by higher-level layers (such as system libraries).

Most systems are heterogeneous, combining the characteristics of more than one architectural style [Garlan and Shaw, 1994]. In this work, however, the pipe-and-filter style will be used (almost) exclusively, as this is a simple abstraction to work with. Although the concepts that will be used will only be applied to the pipe-and-filter style, there is no reasons why they should only be applicable to this style.

Flexibility This section presents the argument that, in order to reduce the costs of software evolution, it is necessary to increase the flexibility of software architecture.

Most effort (in terms of both engineers' time and money) associated with a piece of software is performed after the initial delivery of the system [Lientz and Swanson, 1980, Pigoski, 1996, Banker and Slaughter, 1997]. This effort is necessary because the structure of software is such that it is difficult to modify successfully and correctly.

To reduce the effort spent on software evolution, it is necessary to increase the flexibility of the architecture of software. In many circumstances, it is relatively easy to allow for changes (or categories of changes) which have been anticipated at design-time (for example, inserting modules into an operating system kernel). By contrast, it is much more difficult to allow for arbitrary changes.

One way of achieving a limited degree of flexibility is to parameterise components. For example, the VAT rate used by a component can be changed simply by adjusting one defined constant value. This approach ensures correctness of changes, but is very limited. The only changes that can be made are those that have been foreseen and allowed for at design- and code-times. Most software changes are not foreseen at design-time. For example, if a company was to begin trading in more than one country, and thus had to take account of varying sales taxation rates and methods of indirect taxation (for instance, some countries enforce both local and national sales taxes, and in many countries different categories of goods have different rates of taxation applicable), the simple parameterisation of a single rate of sales tax

would not be sufficient.

Completely unlimited changes can, in principle, be made to any software artefact for which the source code is available. This approach—the opposite of parameterisation in some senses—sacrifices the guarantees of correctness for unbounded flexibility. This is insufficient for successful evolution to occur. Constraints must be made that will allow correct changes to be made, but disallow incorrect changes.

To summarise, then, there are several ways in which a software system can be made flexible. These can be divided into two categories, according to whether they provide flexibility statically (*i.e.*, before run-time) or dynamically (*i.e.*, at run time):

- Compile-time flexibility
  - Compile-time parameterisation (*e.g.*, using defined constants).
  - Composition of components using language constructs.
- Run-time flexibility
  - Run-time parameterisation. For example, the Linux operating system kernel allows run-time parameter setting using the `sysctl` interface [Rubini, 1997].
  - Composition using dynamic binding.

Flexible Architectures A flexible architecture can be defined as an architecture that allows the software elements—from which it is composed—to

be considered, manipulated, removed, and added individually, without damaging the overall properties of the architecture.A flexible architecture must remain flexible under evolution. This flexibility must be maintained by deliberate actions; it cannot be assumed to remain constant during changes to the system [Lehman, 1996]. Flexibility implies simplicity at some level. A piece of software in which the interactions are complex and tangled does not allow modifications to be made easily; a 'clean'design is essential if changes are to be made easily. The concept of architectural flexibility is explored in section 4.2.2.

3.3.1.2 Problems

There has been much work on software architectures. Most of this work has concentrated on forward development of software systems. The main concern of this work is the behaviour and structure of software that has or will change. Change is an intrinsic part of all software systems. To ignore the necessity of change in a software system is to produce difficult-to-modify systems which quickly fall out of use.

When software is changed without specific attention being paid to controlling complexity, its structure degrades [Lehman and Belady, 1985a]. A more desirable outcome is that a software system can be adapted without increasing its complexity.

Current architectural implementations, such as CORBA, lack the ability to introduce new components safely at run-time, without imposing overheads on the implementor, who must ensure that proposed changes are safe, desirable, and so on. Run-time introduction of components is limited to

components which are known at compile-time.

3.3.1.3 Motivation and Examples

This section gives examples of architectural flexibility that have been achieved, and situations where additional architectural flexibility would be advantageous.

Flexibility Use of the C2 architectural style [Oreizy et al., 1998b] has been claimed to ease run-time software evolution; evolution without re-compilation of the system, in such a way that the system retains its integrity without becoming more and more brittle and resistant to change after each modification [Oreizy and Medvidovic, 1998]. Two types of system change are identified: changes to the system requirements, and changes to the implementation that do not affect the requirements.

The architecture of a software system can be used to describe, reason about, and understand the behaviour of a system. Modifications are expressed in terms of the architectural model of a system. There are three types of modification which can be handled by the framework; adding, removing and replacing components. Examples have shown that this is possible [Oreizy and Medvidovic, 1998].

When any form of architectural change occurs, it is important that the mechanics of the change are considered separately from the semantic effects of the change, and that changes must be reasoned about in order to verify that their effects on the system are exactly those which are required [Oreizy and Medvidovic, 1998].

Modern operating systems (such as Sun Solaris and Linux) allow (automatic or manual) insertion of kernel modules at run-time, checking certain compatibility factors (such as version numbers, availability of other required modules) at insertion time. This is not true architectural flexibility, as the only available operations are inserting and removing modules, while reconfiguration of the entire architecture of these operating systems is impossible.

Problems All software which performs useful tasks in an environment must be changed over its lifetime in order to satisfy the requirements of its users [Lehman, 1996]. Typically 50–70% of the effort and expense associated with a piece of software is associated with managing or performing evolution [Lientz and Swanson, 1980]. Examples include business information systems in which the structure of the software should reflect the structure of the organisation, and banking systems, which, in the 1980s and 1990s started to restructure their information systems from an account-based to a customer-based style.

3.3.1.4 Criteria For Success

Success will have been achieved in this area when an architectural style and tools to support the implementation of the style have been defined. The tool support must enable architectural evolution of systems defined in the given style. The tools must support clean abstraction, maintaining the separation of concerns between distinct components, allow reasoning about individual sub-systems, and showing dependencies between individual software entities that make up a system.

3.3.2 Higher-Level Intervention

3.3.2.1 Definitions

Abstraction and Consistency There are many kinds of entity involved in software systems, from the highest level (requirements documentation) to the very lowest (executables). The aim here is to reinforce and—to some degree—automate the correspondence between entities at neighbouring levels of abstraction. In particular, the architecture of a system is not automatically compared against the implementation of the system, so 'creep' can occur: the architectural models which have been carefully generated are inconsistent with the actual implementation.

Components and Connectors In the context of a software system intervening at the architectural level means operating on components and connectors. At this level, it is mainly larger-scale components which are of interest, rather than smaller-scale components (such as standard libraries).

3.3.2.2 Problems

In order to achieve higher-level intervention, the following problems must be addressed:

Abstraction Intervention must take place at a particular level of abstraction. Spanning different levels of abstraction breaks this, and must be avoided. In order to tackle this, concerns must be adequately separated.

Separation Entities which are to be modified must be clearly separated

from the other entities in the system, in order to allow intervention to take place at exactly one place.

3.3.2.3 Motivation and Examples

Modifying software at the code level is too low-level to make changes quickly enough and to ensure that changes are made safely. In order to ensure that component-level safety and functionality constraints are met, intervention at the component level is essential.

To enable component-level intervention, the following properties of the underlying architecture are required:

- A protocol for maintaining information about the components and connectors in the system.
- Change application policies, governing how replacement of components is handled. For example, components can be instantaneously replaced, with all present connections moved to the new component, or old connections can be left in place, with the old component eventually being removed when all connections are closed [Oreizy and Taylor, 1998a].
- A mechanism for interacting with the system at the component level (a maintainers' interface, as opposed to the users' interface).
- Mechanisms for adding, removing, and updating components and the interactions between them. These interactions must happen at the architectural level.

Safe modification is desirable in any system where change must take place, and safe run-time adaption (without affecting availability) is especially necessary in high-availability or safety-critical systems, such as banking systems, power stations, aircraft, *etc.* [Oreizy and Taylor, 1998b].

3.3.2.4 Criteria For Success

Success will have been achieved in this area when a tool has been defined that, in accordance with section 3.3.1.4, allows modification to a system to occur at the software component level, giving software engineers the ability to insert, remove, and update software components, without needing access to the underlying implementation details of the individual components.

3.3.3 Safe Modification

3.3.3.1 Definition

When a system is modified, the modification can be either *safe* or *unsafe.* An unsafe modification is one which results in incorrect behaviour or in premature termination of a program.

3.3.3.2 Problems

In general, it is not possible to simply insert a new component into a system and to be certain of the impact that the insertion will have. Knowledge about the interfaces of the component and implementation assumptions regarding the environment is necessary in order to make this kind of prediction.

In order to tackle the problem of safely modifying a system, then, it is

necessary to categorise the information that is needed to verify the change. The main problem here is specifying without over-specifying.

3.3.3.3 Motivation and Examples

When changes are made, at any level, the maintainers must have justified confidence in the behaviour during and after the change process. For this reason, changes at the architectural level must have predictable and bounded impact on the system as a whole. There must, therefore, be a well-quantified process for making changes to the system, along with a method for determining the impact of proposed changes to a system. Safe modifications are those modifications which do not cause undesired behaviour which was not previously present in a system. Unsafe modification may, for example, cause:

- Premature termination (for any a variety of reasons, including memory access errors, deadlock, *etc.*).
- Incorrect behaviour in components which have previously behaved correctly.
- Incorrect communication; *e.g.,* connecting components which should not be connected, leading to incorrect behaviour.

There are two particular categories of change which can be examined: changes which are made to a system which do not affect the architecture of the system (for example, changes to a component which do not alter its interface, such as changing the type of a private data structure), and changes which do (re-configuration). Changes which do not affect the architecture

are, in general, less complicated to carry out, as there is no ripple effect. Architectural changes are complicated by the fact that, often, there is no up-to-date documentation of the architecture of a system.

3.3.3.4 Criteria For Success

Success will have been achieved in this area when the architectural support tools mentioned in sections 3.3.1.4 and 3.3.2.4 detect and disallow some unsafe modifications to a given software system, as demonstrated in section 3.3.1.1.

3.3.4 Architectural Visibility

It is necessary, in order to successfully understand and modify a software system, to make visible its architecture [Bass et al., 1998]. If changes are to be made at the architectural level, the architecture needs to be made visible.

3.3.4.1 Definitions

Run-Time Visibility The structure of a software system is visible at run-time if it can be determined automatically (*i.e.*, correctly and without human aid) at run-time. There are many ways in which this information can be presented, including diagrams, text in some architectural description language, and so on.

3.3.4.2 Motivation and Examples

If changes to a software system are to be made correctly, those who are performing the changes need to know the state of the system before the

changes are made.

Software systems must be represented in a manner which allows the change to be made at the architectural level. This depiction must include explicit representation of connectors as well as components [Oreizy et al., 1998b].

3.3.4.3 Problems

There are two methods of determining the architecture of a system. The first is to extract the architecture from the source (or object) code by analysis, while the second method is to maintain a model of the system continuously, and to simply make this model visible when requested. Extracting the architecture from the source of a system is a complex task, while automatically maintaining a model of the architecture of a system requires an interface for constructing software which allows the run-time support system to enforce the relationship between the model and the system.

3.3.4.4 Criteria For Success

Success in this area will have been achieved when a method for constructing software which allows the maintenance or extraction of a model at run-time has been developed. The model of the architecture of a running system must be automatically generated (either dynamically when required, or dynamically as the software is modified), accurate, and useful.

3.4 Summary

This chapter has identified the problems that will be addressed in the remainder of this thesis. To summarise, these problems, and their respective criteria for successful solutions, are: *architectural flexibility*, *higher-level intervention*, *safe changes*, and *architectural visibility*.

The following chapters will describe the methods used to solve these problems, the results and the conclusions of the research.

Chapter 4

Research Methods and Techniques

4.1 Introduction

This chapter describes the methods used to undertake the research. The case-studies are explained, as is the form of the results obtained.

In order to provide solutions to the problems identified in chapter 3, a framework for modelling an implementing software in terms of its architecture was developed. In this chapter, the framework is introduced and outlined.

The first part of this chapter deals with the methods that were used, while the remainder shows how these methods were used to develop a framework for creating and evolving software systems. The primary research tool was the building of a framework to enable architectural-level evolution at run-time. This framework was evaluated with respect to several case-studies. The evolution of the framework was then guided by the results of these case

studies.

4.2 Concepts

This section details the key concepts which have been used throughout the research. The terms used are 'framework', 'interaction', and 'flexibility'. The experimental methodology is also described (in section 4.3).

The term 'framework' as it will be used throughout the rest of this document is defined and explored in section 4.2.1, while interaction is covered in section 4.2.3.

One of the most important objectives of the research presented here is providing *architectural flexibility.* The meaning of this term is explored in section 4.2.2

A key assumption made during the development of the framework was that the interactions between components in a piece of software are as important as the components themselves for the purposes of understanding and changing the software. This idea and approach is expanded in section 4.2.3.

The framework incorporates a reflective layer in order that it can represent and control a system. Reflection, which is detailed in section 2.4, is a very powerful tool that can be used to provide visibility and control of the structure of a software system.

4.2.1 Frameworks

Object-oriented application frameworks are an approach to the building of flexible and reusable component systems [Ribeiro-Justo and Cunha, 1999].

Frameworks capture properties of one or more of the domains in which the software is situated [Beck and Johnson, 1994, Fayad and Schmidt, 1997] (such as application domains, or architectural styles) by providing reusable components, either for application domain objects, or for architectural entities. In this document, architectural frameworks are of interest.

Architectural frameworks allow easy construction of software systems from individual components, by standardising the interface schema and encouraging design reuse [Johnson, 1997]. A framework dictates the architecture of systems which are built with it, and encapsulates certain design decisions (made by the framework designer) [Gamma et al., 1995]. Use of application frameworks enables more rapid construction of software, more effective reuse of code, encourages design reuse [Gamma et al., 1995], and is useful in product-line architectures.

4.2.2 Architectural Flexibility

In order to allow successful software evolution, flexible architectures are required. A flexible architecture is more than cleanly constructed and well understood; the effort required to make changes to the system must be proportional to the size of the change.

In order to quickly and easily modify a software artefact, the ability to add, remove, and update the components and connectors that make up the artefact is necessary. This gives the software maintainer—the person or people responsible for making changes to the software—the ability to introduce a new component (for example, to introduce new functionality into

the system) at run-time, without compromising the integrity of the original parts of the system.

There is a spectrum of architectural flexibility. At the least flexible end of the spectrum, the architecture of a system is completely rigid; the components within the architecture can be replaced, but the interactions between the components are fixed and static. At the other extreme, the architecture is completely flexible; components and connectors can both be added, removed, and replaced. The least flexible option is too restrictive to allow many kinds of modification. For example, under these restrictions, introducing a new kind of file system to an operating system would be impossible. At the opposite extreme, complete flexibility is obtained, at the expense of safety.

The safest starting point is the least flexible. If the architecture of a system cannot be changed, the properties of the architecture can be guaranteed to remain constant between versions. However, all (E-type) software must change. Often this requires changes at the architectural level. Hence, the totally safe option is too inflexible. The totally flexible end of the spectrum is also unsuitable, as it does not prevent unsafe and unsound modifications. The spectrum of architectural flexibility can be broken down into the following categories, in increasing order of flexibility and decreasing order of safety:

Complete Rigidity No run-time changes can be made to the system. At this point, there is no need to make any explicit statements about the architecture at run-time.

Architectural Rigidity Components can be interchanged, although this is only possible when interfaces are preserved. The connectors cannot be changed in any way. Complete safety of the architecture is assured (although faulty components can still be introduced). At this stage, the set of components needs to be visible (in some manner) at run-time, but the connectors need not be. No further architectural information is required at run-time.

Invariant Flexibility Components can be replaced, preserving interfaces, and new components and connectors can be introduced in a manner that has been pre-determined. For example, introducing a new pump into a petrol station system is both valid and safe. The set of safe modifications that can be made is identified by a set of invariants which cannot be broken. In the petrol station example, example invariants, expressed informally, would be "each pump must be connected to exactly one central controller" and "there is exactly one central controller". Here, the architecture of the system must be visible at run-time, although the invariants need not be.

Flexible Invariants The set of architectural invariants is made explicit and modifiable at run-time. In this case, the architectural properties of the system can be broken between versions of the software, though this must be done in a conscious manner by maintainers.

Architectural Flexibility Components can be interchanged, even if interfaces are not preserved. Connector configuration can be changed in any way. Invariants are not used or maintained. Some architectural

information must be made explicit at run-time in order to allow modification, though invariants are not used, as safety of modifications is not considered. No properties of the system need be preserved between versions.

An architecture which is flexible *at run-time* will provide the following benefits to software engineering:

- Continuous availability [Oreizy and Medvidovic, 1998].
- Controlled change [Oreizy and Taylor, 1998a].
- Change at a level which reflects the level at which the system is understood by the designers [Perry and Wolf, 1992].

4.2.3 Interaction

Any software system consists of a set of components (which provide functionality), interacting *via* a set of connectors (which provide a communication infrastructure for the software). Software engineering has, for most of its history, concentrated on the entities which make up software systems, regarding the means by which they interact as second-class citizens. By contrast, in the framework that is described here, the connectors by which components interact are considered of primary importance.

As described in section 3.3.1.1, the architecture of a software system is composed of a set of components and a set of connectors. Interactions within a system are both captured and modelled using the connectors. Thus, a model of a system (the 'architecture') contains representations of the same

components that are used in the implementation of the system. In this way, the problem of system documentation becoming out-of-date with respect to the system is addressed: the system *is* (a part of) its own documentation.

Traditionally, software engineering has concentrated almost exclusively on the entities of software systems. Recently, however, there has been much work which has considered interaction as of at least equal importance as the behavioural entities [Shaw and Garlan, 1996, Shaw, 1993]. This work has recognised the traditional deficiency in system 'architecture' diagrams and descriptions with respect to the semantics attached (or otherwise) to the 'lines' which connect design elements. Although connectors are ultimately implemented in terms of a small set of elementary constructs (principally procedure call and shared memory, also various networking primitives) [Shaw, 1993], there are a wide range of higher-level connector types, such as pipes, call-and-return, implicit invocation [Shaw and Garlan, 1996] which generalise this. Consideration of these connectors as first-class entities in their own right is important in order to capture design decisions with respect to interactions, distribution [Shaw, 1993], to support evolution [Oreizy et al., 1998b], and to help avoid problems in composing components with incompatible interfaces, by making the properties of the connectors which connect these interfaces explicit [Garlan et al., 1995].

4.3 Experimental Approach

4.3.1 Outline of Approach

In order to address the problems identified in chapter 3, a framework for modelling software systems and their evolution was created. This framework was motivated by a set of case studies, described in section 4.5. The framework allows the modelling and construction of the system in question, and the run-time manipulation of its architecture. This section describes the case-study-based approach, and gives the details of each of the case studies.

In common with most frameworks, components must conform to a specific interface (described in section 4.4.1.1), and certain assumptions about their behaviour are made. These assumptions are identified in section 4.3.2

For each case-study, an implementation was built, using the framework. In each case, this implementation motivated extensions and changes to the framework. As the case-studies progressed, the framework was continuously evaluated and modified. This approach thus lead to a framework which is applicable to each of the case-studies.

The initial framework created was designed to have the following features: the ability to modify the system at run-time; an explicit run-time representation of components and connectors; disallowing some kinds of unsafe change; and run-time visibility of the architectural structure of a system.

Using the case-study based approach allows the development of a framework which addresses the problems identified during the implementation of solutions to real (albeit small) programming problems.

4.3.2 Assumptions

In order to successfully implement a case-study in the framework, the example must fulfil certain expectations. These expectations are outlined in this section. There are two main types of assumption, regarding the architectural and behavioural properties of systems.

The framework models and manages systems which conform to the pipe and filter architectural style. Any system which does not do this cannot be modelled without modification.

The systems under consideration must contain components which match the given interface, and only communicate with other components using facilities provided by the framework (*i.e.*, using message-passing through pipes). Although it is possible for components to communicate other than *via* pipes (*e.g.*, using external files), this breaks the model of architecture and can lead to undesirable behaviour when modifications are made.

4.3.2.1 Architectural Assumptions

A framework cannot cover all possible architectural styles. In order to sensibly limit the scope of the framework, it was decided to concentrate on pipe-and-filter style systems (described in section 2.3.2.1), as these have well-understood modelling techniques and languages associated with them. It was assumed that each component has a set of input ports and a set of output ports, and that no port is bi-directional. The framework also makes the assumption that inter-component communication is solely through the mechanisms provided (namely pipes), and that components do not use any other

form of message-passing or shared memory. This assumption is not enforced, and in fact components can communicate outside the system, but doing this invalidates the architectural model of the system that is maintained by the framework.

4.3.2.2 Evolutionary Assumptions

It was assumed that all evolution will leave the architectural style intact; dynamically changing from a pipe-and-filter style to a layered architecture is not feasible, for instance. It was also assumed that each change will be either a localised change of or to a component, or that each change can be decomposed into changes of this type.

4.3.2.3 Modelling and Implementation

Each case-study must be implemented using the facilities provided by the framework. In order for this to be done successfully, the system under consideration must be modelled using the pipe-and-filter style, and implemented using components which conform to the `Component` interface. Modelling of the system can be performed in any language (formal or informal) which supports the pipe-and-filter style. In principle any suitable notation can be used. In this work, the Darwin ADL was used for architectural description.

When the system has been modelled, it must be implemented. This involves creating a set of classes which extend the `Filter` abstract class. In the examples studied here, and in most other examples, it is not necessary to extend the `Pipe` class.

4.3.2.4 Evolution

In order to determine how well the framework supports the evolution of systems that it is modelling, it is necessary to make changes to these systems. This is one of the key parts of the case-study.

The kind of changes which are made are dependent on the nature of the system under consideration. The framework is designed to manage and support architectural evolution, so this kind of change is the most suitable. For example, it is often useful to replace a component in the system in order to correct errors or to experiment with different behaviour (such as modifying a component to evaluate different algorithms).

There are many ways in which changes can be carried out. The main way in which the framework supports change is through allowing architectural reconfiguration. For example, it is possible to modify the architecture of a system in several ways, including adding and removing both components and connectors.

4.3.2.5 Evaluation

Having carried out the case study, it is necessary to determine how well the framework supported the process of modelling, implementing, and changing the system. This is an important stage in motivating future developments of the framework.

The criteria used to evaluate the framework in the light of a case study are:

- How well did the framework support the initial modelling of the system?

In some cases, a mismatch may be due to the choice of an inappropriate case study (such as an event-driven programming language interpreter), rather than due to failings in the framework. Alternatively, they may be due to deficiencies in the framework's implementation of the style (initially, the framework only supported a single input and a single output connection to each filter, which was a limitation in the implementation of the style, not in the style itself).

- Once the system had been modelling in the pipe-and-filter style, was its implementation straightforward? Difficulties in this stage are more likely due to failures in the framework than problems with the case study. Problems at this stage usually result from a mismatch between the pipe-and-filter style and the framework's implementation of it.

- Was the evolution of the system possible? How well did the framework support it? At this stage, it is useful to note steps which could have been (but were not) automatic. Problems here can be caused by style mismatches (*e.g.*, trying to make a change that results in a departure from a pure pipe-and-filter style).

4.4 Research Methods

The research presented here proceeded by means of building a framework to allow evolution at the architectural level. This framework was then evaluated with respect to a set of case studies. The framework was then modified to accommodate new features that were determined to be necessary to success-

fully handle the case studies.

4.4.1 The Framework

This section discusses the framework that was built as part of the research. The design, implementation, and evolution of the framework are described.

4.4.1.1 Design

The aims of the framework, related to the objectives described in section 3.4, are to:

Architectural Flexibility Allow the composition and evolution of software systems at the architectural level of abstraction.

Higher-Level Intervention Maintain the consistency of software systems over evolutions.

Safe Changes Ensuring that only safe changes are made to a system.

Architectural Visibility Provide visibility and control of software systems at run-time.

The use of a reflective layer, to maintain and control meta-level information, addresses these points. Allowing visibility and manipulation of meta-level information (properties of interactions) gives the maintainer of software the ability to compose software at the architectural level.

Overview A software system is composed of two levels of entity; base-level entities and meta-level entities. In the framework that is described here, base-level entities are components in the software system that is being modelled and controlled, and meta-level entities control the interactions between the base-level entities. The main components which make up the framework can be seen in figure 4.1.

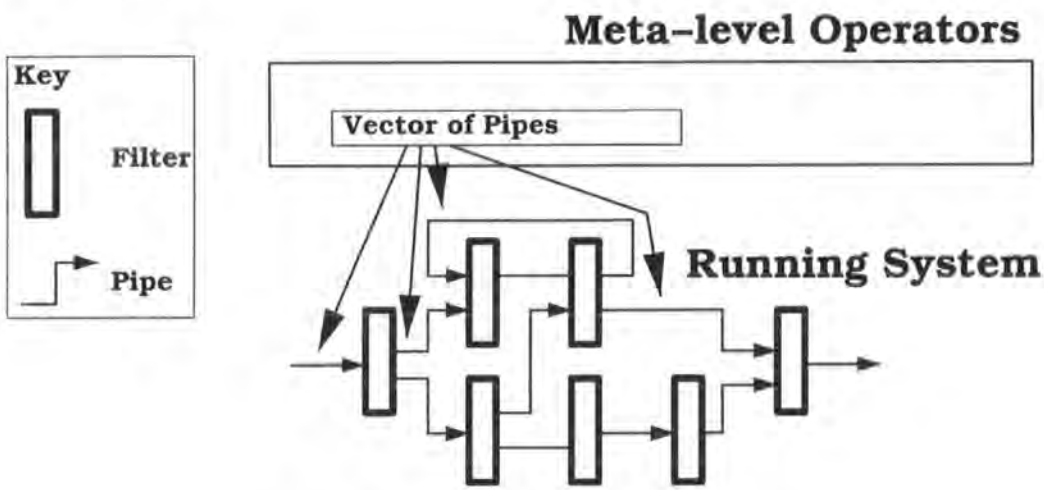

Figure 4.1: Overview of the Framework

System Architecture The main components of a software system which has been created using the framework are:

- A set of pipes, representing the interaction between components. Pipes are one-way, asynchronous, data streams between exactly two components.

- A set of filters, representing the functionality of software. Filters have a finite (possibly zero) number of input and output ports by which data enters and leaves respectively.

There are two principle ways in which pipes and filters differ. Pipes have exactly one filter connected at each end, and make no change to the data which passes through them. By contrast, filters can perform operations on the data which passes through them (and need not produce exactly one output message for every input message, unlike pipes), and can be connected to more than one pipe.

The framework supports these classes of component by maintaining a set of 'pipe' objects which hold information about the interactions between the components. Each 'pipe' object maintains information about the filters involved in the interaction, the ports which are used, *etc.*

Component Design The `Pipe` component type is fixed. `Pipes` have the interface shown in section A.1.

The `Pipe` component is conceptually very simple: messages are received at the input end, and transmitted from the output end. A `Pipe` recieves input by polling the component connected to its input (using the `hasMessages()` method for the relevant port). Messages are then sent by the pipe to the output filter (after polling the `Filter.isReadyToReceive()` method) using the `Filter.receive()` method. Internally, each `Pipe` maintains a FIFO buffer of messages. `Pipes` are implemented as Java threads, in order that multiple instances can coexist in a system.

In order to allow the framework to control and monitor the set of `Pipes` which it maintains, there are various methods. These are:

`getInput()` and `getOutput()` Return the `Filters` that are connected to each end of the `Pipe`. These provide part of the reflective capabilities

of the framwork, and allow the framework to determine the architecture of the system which it is controlling.

`pauseInput()` **and** `pauseOutput()` Are necessary to allow the framework to suspend input to or output from a `Pipe` which is being modified in some way (for example, if the input filter is to be removed or replaced). The methods `resumeInput()` and `resumeOutput()` are used to restart processing by the `Pipe`.

`setOutputDestination()` **and** `setInputSource()` Are reflective operations, used by the framework to modify the architecture of the system which it is controlling.

`addPipe()` Used when two `Pipes` are to be merged.

`run()` **and** `terminate()` Used by the framework to start and stop a `Pipe`.

Filters, on the other hand, are specific to particular functionality. The interface is shown in section A.2. Particular components conform to the `Filter` interface, while providing their own functionality.

While `Pipes` are maintained and implemented exclusively by the framework, `Filters` are at the interface between the framework and systems which are implemented using it: a programmer must create specialised objects of type `Filter`, which contain features relevant to the domain of the software in question as well as the necessary features required for the framework. Thus a programmer creating a system for implementation in the framework will not have to produce any objects of type `Pipe`, they will be required to implement objects of type `Filter`, and produce code for each of the methods

listed below.

The two principle methods on `Filter` relating to inter-component communication are:

`send()` Called by a `Pipe` to pass a message (of type `Object`) to the `Filter`. This method is only called after a call to `isReadyToReceive()` on the `Filter` has returned `true`, indicating that the `Filter` is prepared to accept a message. If a call to `isReadyToReceive()` returns `false`, the message is buffered in the `Pipe`. `Pipes` which have messages poll their output `Filters` until each message has been sent. In this way, a `Pipe` acts as a buffer.

`receive()` Called by a `Pipe` to receive a message from the `Filter`. This method is only called after a call to `hasMessages()` has returned `true`. Similarly to the `send()` protocol, a `Pipe` will poll its input filter.

When a change involving adding a `Pipe` to a `Filter` is being made, the framework will call either of the methods `canActivateInputPort()` or `canActivateOutputPort` in order to determine whether the `Filter` is capable of accepting a `Pipe` attachment to the relevant port. If this call returns a value of `true`, the pipe is connected and either `activateInputPort()` or `activateOutputPort()` is called (depending on the direction of the port in question). In this way, the `Filter` is made aware of connections being made to it, and thus can take appropriate actions.

If a `Filter` is removed from the system, the `terminate()` method is called, which should perform any deallocation or other tidying which is required.

4.5 Case Studies

In this section, the case studies are introduced. Each case study was chosen because it motivates or exercises features of the framework. The first case-study (KWIC) is the simplest, and was intended to motivate the simple, initial, features of the framework. The Markov-chain text generator was the next case-study, and was more complex. This case-study motivated more advanced features, while verifying those created during the process of implementing the first case-study. The final case-study (the Gas Station) is the most complex of the three, and involved a more sophisticated architecture than the previous two examples.

Each case study had some common elements; the architecture of the system was specified in Darwin, and the system was implemented in the Java programming language.

The case studies were selected because they have the following characteristics:

Modularity Each example can be divided into separate and well-defined modules.

Abstraction The key components of each example are at a sufficiently high-level of abstraction that a boundary can be drawn between the architecture of the system and its behaviour.

Well-known The various case-studies have been studied in the literature, and are well-understood.

Varying Size The size and level of abstraction of the examples varies, al-

lowing comparisons of the level of effectiveness of the experimental approach to be made.

4.5.1 Key Word In Context

4.5.1.1 Introduction

The Key Word In Context program (KWIC) [Parnas, 1972] creates a permuted index of a document. The program takes a text file consisting of a set of zero or more lines each of which are composed of zero or more characters. A permuted index, consisting of a set of numbered lines, sorted into alphabetical order, showing the context of each word in the input text file, is produced.

The KWIC case-study was chosen as it is a simple and well-understood problem with a well-formed architecture [Parnas, 1972].

The architecture of the KWIC case-study is shown in figure 4.2.

The KWIC case-study was the first to be implemented, and thus motivated several of the basic features of the architectural framework.

The initial framework created to model and implement the KWIC case-study was designed to have the following features:

Modifiability of the system at run-time to fulfil one of the basic aims of this research.

High-level modification of the architecture of the system.

Disallowing some kinds of unsafe change by only allowing a given set of changes to be made within the system. These changes are: adding

and removing components; adding connectors to and removing them from components. It is only possible to add a connector to a component if that component is willing to accept it.

Explicit run-time representation of components and connectors in order to show the architecture of the system and to allow its modification.

4.5.1.2 Assumptions

The KWIC case-study partly motivated some of the assumptions made by the framework, so to a certain extent, it matches the framework by default. The case-study was implemented using four classes, as described in section 4.5.1. Each class in the system is a single-input, single-output filter, making it suitable for a pipe-and-filter architectural model.

4.5.1.3 Modelling and Implementation

The architecture of the KWIC system is given informally in figure 4.3. A more precise way of specifying the architecture is to represent the architecture in Darwin, as shown in figure 4.2.

The KWIC system was implemented in Java, using the `Filter` class (described in section 4.4.1.1). Initially, the system was modelled using a simpler model of `Pipes`, with each pipe having only a single input and a single output port. This modelling restriction was lifted later on, when other case studies made it unrealistic.

There are four components in the KWIC system. They are as follows:

Input Opens a file, reads it line-by-line, and passes each (num-

```
interface NL {} // numbered lines from a file
interface NSL {} // numbered, shifted lines

component Input {
   provide numberedLines:NL;
}

component CircularShifter {
   require numberedLines:NL;
   provide numberedShiftedLines:NSL;
}

component Sorter {
   require numberedShiftedLines:NSL;
   provide sortedNumberedShiftedLines:NSL; // in order
}

component Output {
   require sortedNumberedShiftedLines:NSL;
}

component System {
   inst
      I: Input;
      C: CircularShifter;
      S: Sorter;
      O: Output;
   bind
      I.numberedLines -- C.numberedLines;
      C.numberedShiftedLines -- S.numberedShiftedLines;
      S.sortedNumberedShiftedLines --
                           O.sortedNumberedShiftedLines;
}
```

Figure 4.2: The KWIC System specified in Darwin

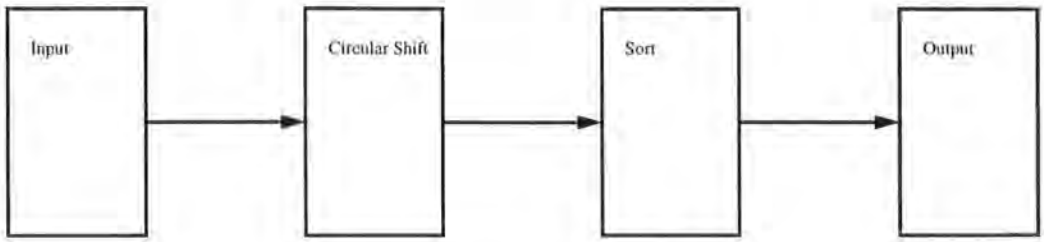

Figure 4.3: Architecture diagram for KWIC

bered) line to the next component.

Circular Shift Rotates each line, to produce a set of shifted, numbered lines.

Sort Sorts the set of lines produced by the previous component into alphabetical order of first word.

Output Displays the sorted, numbered lines produced by the previous component, in order and formatted for easy reading.

The input component queries the user of the system for a file. This file is opened and each line is passed (along with its number in the file) into the next filter. The circular shift filter takes a numbered line at a time, and produces all the circular shifts of that line. For example, the line `("The quick brown fox", 1)` would be rendered as the four numbered strings

1. `("", "The quick brown fox", 1)`
2. `("The", "quick brown fox", 1)`
3. `("The quick", "brown fox", 1)`
4. `("The quick brown", "fox", 1)`

The tuple produced by the shifting component has the form (*beginning of line*, *end of line*, *line number*). The end of the line forms the key by which the lines are sorted, therefore this part of the tuple cannot be an empty string.

The sort component takes each of these tuples and sorts them into alphabetical order of the second string. In the example, this order is 3, 4, 2, 1.

4.5.1.4 Evolution

The KWIC case-study was carried out in order to motivate and exercise the modelling and implementation capabilities of the framework, so the evolutionary aspects of the framework were not as well tested as in the more sophisticated cases. The kinds of changes which were made were simple addition of components. For example, monitoring `Filters` (which simply show all data passing through them in an on-screen window) were used in order to show that `Pipes` can be interrupted by the addition of new components and the system as a whole will still function correctly once the new component has been introduced and each `Pipe` connected.

4.5.1.5 Evaluation

This was the first case-study, and as such it motivated many of the features of the framework. In this case, the framework supported the modelling and evolution of the KWIC case study well, as the case-study was used to motivate the creation of the framework.

4.5.2 Markov-Chain Random Text Generator

4.5.2.1 Introduction

The Markov-chain random text generator takes a piece of text, and produces a statistical model of the language used in it. This model is then used to generate a random text whose language is similar in style to the original [Kernighan and Pike, 1999].

4.5.2.2 Assumptions

The Markov-chain text generator fits the pipe-and-filter style with some caveats. At some points in the design, the case-study would be improved if the simple data-flow model of the framework was augmented with a shared-memory artefact (as used in, for example, blackboard and other architectural styles [Garlan and Shaw, 1994]), as seen in figure 4.4. This would avoid inefficiencies in passing large amounts of data (in this case a large hash table) through the system unaltered. Using a blackboard to store the language model would also reduce coupling in the design; the technology used could be changed from a hash table to an alternative without having to modify the text generator. Apart from this limitation, the case-study fits the pipe-and-filter architectural model well.

4.5.2.3 Modelling and Implementation

There are four components in the system:

Input This component prompts the user for a text file and outputs the content of that file.

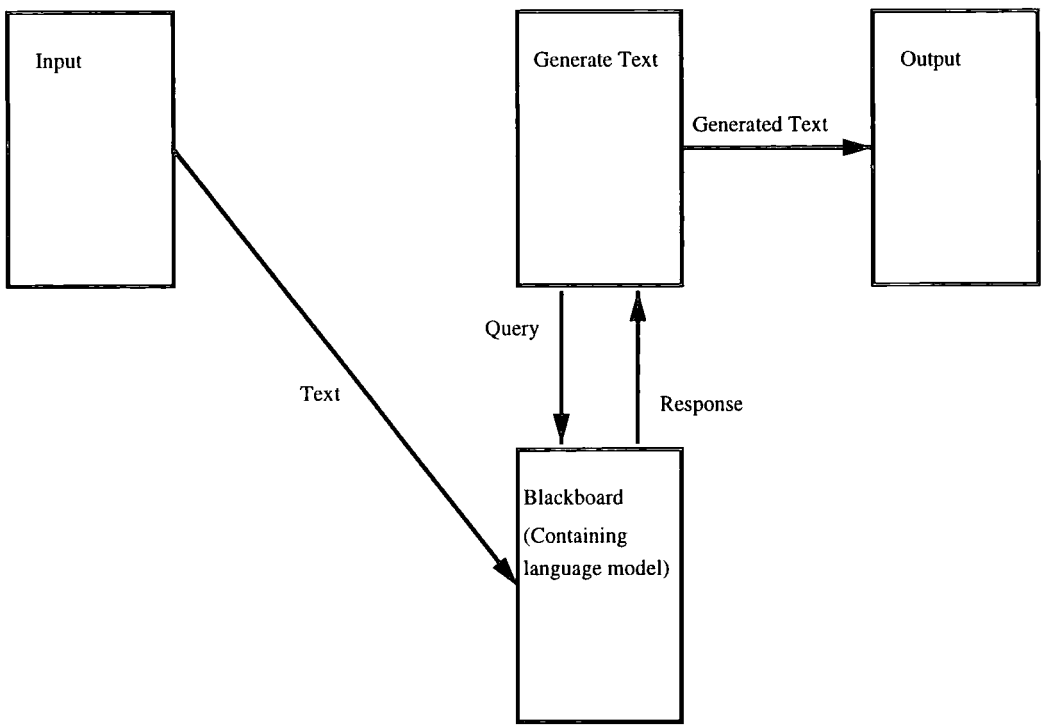

Figure 4.4: An alternative architectural style for the Markov-Chain Text Generator

Hash This component constructs a set of word prefix/suffix pairs which model the language used in the input text.

Generate This component uses the model of language produced by the previous component to generate a new text, which is passed to the next component.

Output This component takes the text produced by the previous component and formats it for display.

The architecture of the system is shown informally in figure 4.5 and given in Darwin in figure 4.6.

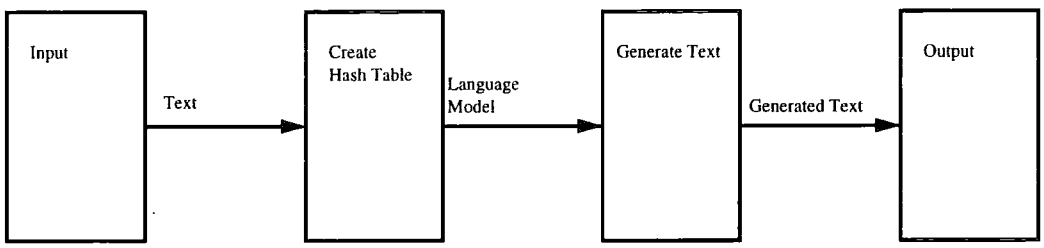

Figure 4.5: Architecture diagram for the Markov text-generator

The input component queries the user of the system for a file, which is then opened and passed to the next filter. The hash component then receives the text and builds a hash table of word prefix/suffix pairs. This hash table is then passed to the generator component, which produces random text to pass to the output component.

As in the previous system, each component has at most one input and one output, with only ports numbered 0 being used.

```
interface Text {} // raw text
interface LanguageModel {} // Model of language,
                           //  prefix/suffixes

component Input {
   provide inputText:Text;
}

component Hash {
   require inputText:Text;
   provide model:LanguageModel;
}

component Generate {
   require model:LanguageModel;
   provide outputText:Text;
}

component Output {
   require outputText:Text;
}

component System {
   inst
      I: Input;
      H: Hash;
      G: Generate;
      O: Output;
   bind
      I.inputText -- H.inputText;
      H.model -- G.model;
      G.outputText -- O.outputText;
}
```

Figure 4.6: The Markov Text Generator specified in Darwin

4.5.2.4 Evolution

As with the KWIC system, the framework supports modelling of the system, and also evolution. The text-generator can be changed, for example, by replacing the hash-table creating component in order to experiment with the efficiency and results of different designs (for example, using character-based rather than word-based processing). The text-generation component can be replaced, in order to experiment with variations on the algorithm (*e.g.*, using a different prefix length when generating text, which involves a simple change to a compile-time constant).

4.5.2.5 Evaluation

As described above, the modelling aspects of the framework suited this example well, though with the following comment: A central blackboard component could, in fact, have been used to make the system more efficient, but this might have involved adding synchronisation capabilities to the system (not necessarily to the framework, however) in order to ensure that data is received in a timely fashion by those components which require it.

4.5.3 Gas Station

4.5.3.1 Introduction

The so-called gas station example[Ducasse and Günter, 1998] involves a cashier component, and a set of one or more pump components. The station is a North American style station; payment is made before fuel is drawn from a pump.

4.5.3.2 Assumptions

The gas station is superficially unsuitable for implementation in the pipe-and-filter style as it seems that the communication between components should not be queued (which is implicit in the style, at least as it is implemented in the framework). However, the Darwin model shown in section 4.8 demonstrates that the gas station can be adequately modelled in this style.

4.5.3.3 Modelling and Implementation

As in all of the case-studies, the gas station was modelled in Darwin and implemented in Java. The need for unqueued messages can be handled by ensuring that the thread in the `Cashier` component which handles the messages from the `Pump` components handles messages in reasonable time. This is not a difficult task to achieve, and the implementation of the `Pump` component ensures that messages are not queued in the pipe between a `Pump` and the `Cashier`.

In order to correctly handle the three `Pipes` required for each `Pump`, each `Pump` uses one output port and two input ports. From the point of view of the `Cashier`, `free()` and `load(amount)` messages to `Pump` $n \geq 0$ are sent from ports $2n$ and $2n+1$ respectively. All `isFree()` messages from this pump are received on port n.

There are two categories of component in the gas station system; the cashier (of which there is always exactly one), and the pumps (of which there is always at least one).

An informal overview of the gas station system is presented in figure 4.7,

while a more rigorous description is given (in Darwin) in figure 4.8. This figure shows the components and connectors involved in the gas station. The components (filters) are represented by boxes, with connectors (pipes) represented as lines, with arrow-heads indicating message flow. In this diagram, there is no 'customer' component indicated: even though this kind of component could be considered part of the system [Ducasse and Günter, 1998], it is not considered so here. Each pump filter is identical in terms of message streams; for simplicity, only one set of these pipes has been labelled.

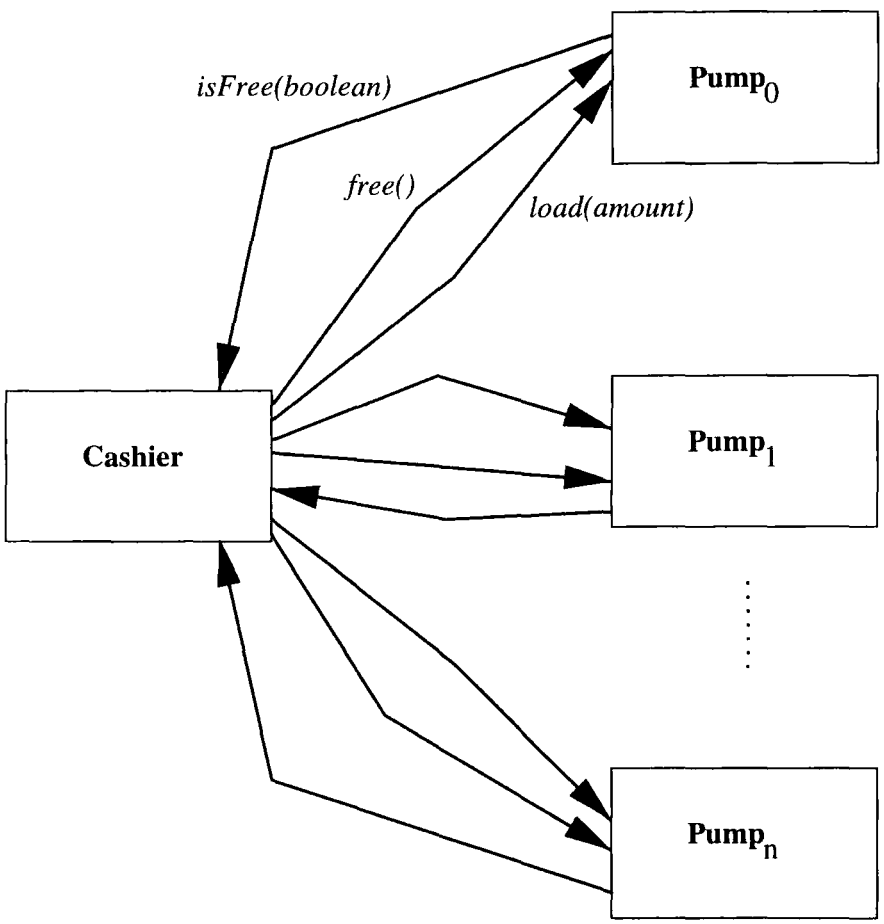

Figure 4.7: Architecture diagram for the Gas Station

```
// Want exactly one of these
component Cashier (int n){
   provide checkFree [0..n];
   provide loadPump [0..n];
   require isFree [0..n];
}


// Want n of these (dynamically added)
component Pump {
   require checkFree;
   require loadPump;

   provide isFree;
}

component System (int n) {
   array pumps[n];

   inst cash:Cashier(n);

   forall i=0 to (n-1) {
      inst pump:Pump;
          bind cash.checkFree[i] -- pump.checkfree;
        }
}
```

Figure 4.8: The Gas Station specified in Darwin

The cashier component is the initial 'point of contact' between the customer and the system. The customer pays the cashier for fuel, the cashier finds an available pump (using the '`free()`' protocol) and then loads this pump with the appropriate amount of fuel.

When a pump receives a '`free()`' query from the cashier, it responds with a boolean. When a pump has been successfully `load()`ed with a given amount of fuel, the customer can then use the pump to obtain the fuel.

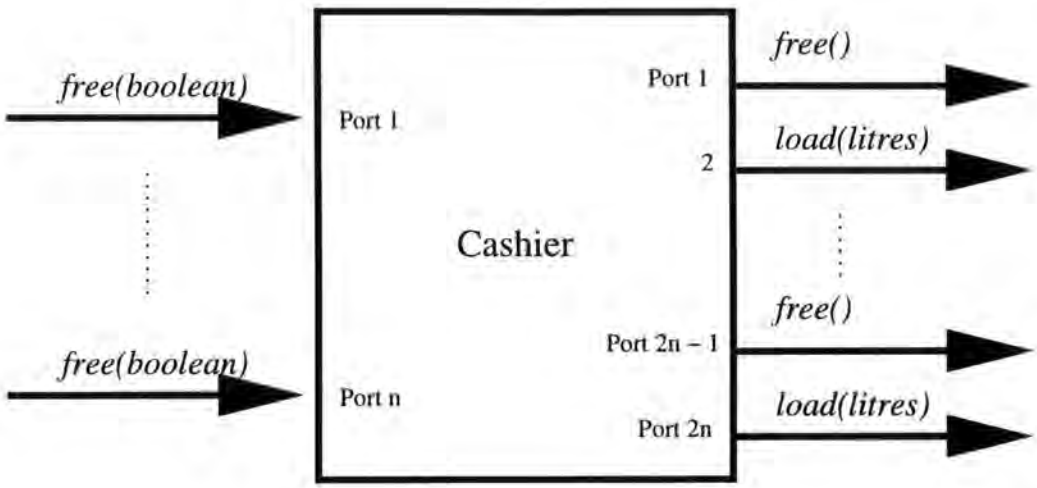

Figure 4.9: 'Cashier' Component

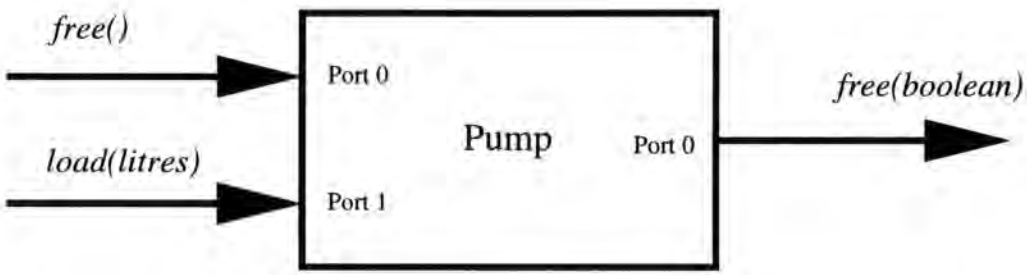

Figure 4.10: 'Pump' Component

When a customer has paid for some fuel, the cashier attempts to find a pump that is free. Each pump in turn is tested using the 'free' message. When a pump eventually returns '`true`' (*via* an '`isFree(boolean)`' message on the corresponding port), that pump is then `load()`ed with the correct amount of fuel.

Each pump has a single output port (number 0), and two input ports (0 and 1). Input port 0 receives '`free()`' query messages from the cashier. When such a message is received, a pump responds immediately[1] with an '`isFree(`{*true,false*}`)`' message on the output port. `load(amount)` messages are received on input port 1, and cause the pump to become ready to dispense a given amount of fuel[2].

There are several questions relating to the representation and implementation of customers:

- Are customers explicitly represented, or treated as part of the 'environment' that the system inhabits?
- If customers are explicitly represented as part of the system, how are they modelled? The approach of inserting a new component at runtime each time a new customer is introduced to the system has some attraction, but also seems to contradict the use of reflective operations to undertake 'normal' operations.
- How does a customer (rather, the *representation* of a customer) determine which pump to use? There are two issues here: how is the

[1] There are no real-time concepts in the system; 'immediately' is a loose term.

[2] In a physical implementation, the pump would have to draw this amount of fuel from the gas station's tanks.

information returned to the customer component, and how then does the component 'find' the appropriate pump component. Dynamically inserting a connector (a pipe) seems contrary to the purpose of reflective operations.

4.5.3.4 Evolution

The most obvious form of evolution of the gas station is to add or remove pumps. This is supported by the framework, as long as the maintainer ensures that the correct pipes are connected to the correct ports (as described in section 4.5.3.3).

There are various ways in which the gas station can evolve. These are divided into two categories; architecture-preserving changes (such as adding or removing pumps) which maintain the structural properties of the system, and architecture-modifying changes, which change the structural properties of the system.

Architecture-preserving properties which are foreseen at design-time are easily accommodated at run-time, as the appropriate structures are in place for the changes to be made. For example, the insertion of pumps at run-time is enabled by allowing the dynamic instantiation of the software component which represents the pump, and the dynamic binding of the connectors in the appropriate fashion (maintaining the architectural invariants).

Architecture-modifying changes are currently impossible to handle at run-time in any significant way. For example, dynamically changing a compiler from a pipeline to a blackboard architecture [Garlan and Shaw, 1994] is not possible.

4.5.3.5 Evaluation

Initially, the single-port restriction on the `Filter` class meant that the implementation of the gas station was impossible. Once this restriction had been lifted, the framework successfully supports the implementation of the system.

4.6 Summary

This chapter has outlined the methods that have been used to undertake the research. The framework that is part of the results has been described, as have the case studies that have been used to guide the evaluation and evolution of the framework.

Chapter 5

Research Process

5.1 Introduction

This chapter describes the process of the research. Section 5.2 describes the development of the framework, and the results of the case-studies are given in section 5.3.

5.2 Framework Development

This section outlines the development of the framework.

5.2.1 Motivation and Objectives

The design and implementation of the framework were motivated by the problems given in chapter 3. The main criteria for the framework, as stated in section 3.4, are *architectural flexibility*, *higher-level intervention*, *safe changes*, and *architectural visibility*.

In order to satisfy these criteria, the framework has the following features:

- A run-time representation of the structure of a software system (modelling the components and the connectors), giving visibility, and addressing *architectural visibility.* This representation is examined in section 5.2.2.6.

- A graphical interface, allowing the insertion and deletion of components and connectors which make up a software system, addressing *architectural flexibility*, and *higher-level intervention.* This interface is detailed in section 5.2.2.8.

- A model of a particular software architectural style, which consists of components which communicate *via* connectors, aiding in addressing *architectural visibility* and *safe changes.* This model of software is detailed in section 5.2.2.

5.2.2 The Pipe and Filter Model of Software

5.2.2.1 Outline of the Model

The pipe-and-filter architectural style employs two categories of architectural entity. The computational components of software are filters, which are connected using pipes. Filters interact *via* a set of input and output ports, which are connected to pipes. Pipes are data streams, transmitting data in first-in first-out order. An example of such a system is shown in figure 5.1. Components perform computation, producing, consuming, or altering messages. Messages are passed along pipes asynchronously (ordered, in respect to first-

in, first-out). Each pipe/filter connection is *via* a particular port. Ports are numbered sequentially from zero.

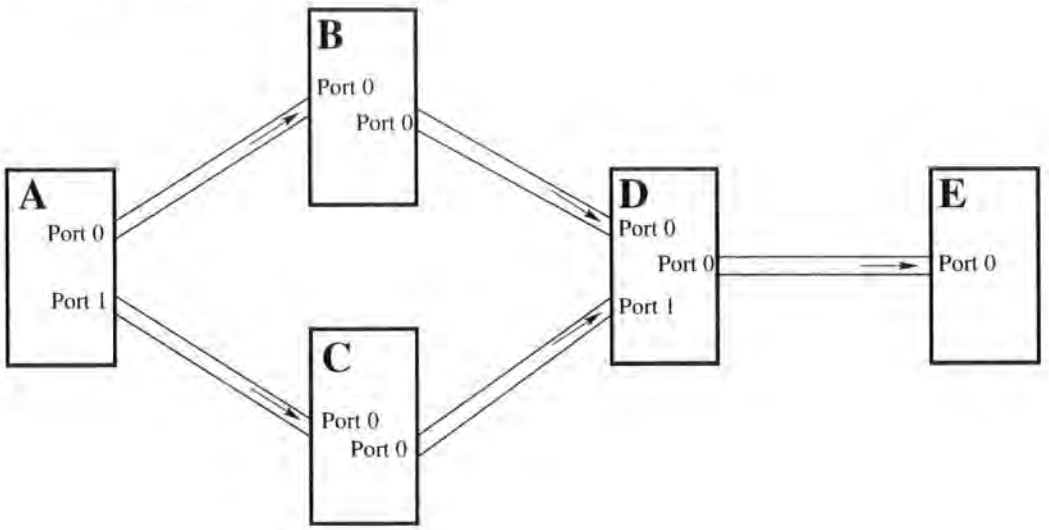

Figure 5.1: An Example Pipe And Filter System

5.2.2.2 Architectural Styles

An architectural style encapsulates constraints on software architectures. A style is determined by the set of component and connector types which are either required or supported, constraints on the connector configuration of a system [Bass et al., 1998]. The main categories of architectural styles are data-centred styles (such as blackboard architectures), data-flow styles (such as the pipe-and-filter style discussed below), virtual-machine styles (such as that used in interpreted languages *e.g.*, Java), call-and-return styles (as used in traditional procedural programming languages), and layered styles (such as those used in operating systems).

5.2.2.3 The Pipe and Filter Style

In the pipe and filter architectural style, the components are filters, and the connectors are pipes. Data flows through the pipes in one direction only, into or out of ports to filters.

5.2.2.4 Implementation of the Style

In the architecture presented here, pipes are implemented by the framework. Pipes consist of asynchronous buffers of objects between filters. Filters have two types of port, input and output, which can only be connected to the corresponding port of a pipe. Each pipe is implemented as a thread, executing independently and polling each filter that it is attached to. Filters implement various methods in order to maintain the relationship with the pipes that they are connected to. These methods (which are called by the pipes) are:

- public boolean hasMessages(int outPortNo); returns true if the filter has any messages ready to be transmitted.
- public Object send(int outPortNo); returns a message to be sent.
- public boolean isReadyToReceive(int inPortNo); returns true if the filter is capable of receiving a message.
- public void receive(int inPortNo,Object message); sends a message *to* the filter (from the pipe).
- public boolean activateInputPort(int portNo); and public boolean activateOutputPort(int portNo); tell the filter that pipes have been connected to the respective ports. Returns true on success, false on failure.

5.2.2.5 Programming Language

The framework has been implemented in Java. This choice was made because the language provides reflective facilities, is interpreted (providing facilities for dynamic loading of class definitions), and provides a rich application programming interface, with facilities for collections, user-interfaces, and other useful capabilities.

Reflection The package `java.lang` provides meta-classes for `Object` and `Class`. Along with classes provided by `java.lang.reflect`, these can be used to inspect and use classes. In conjunction with the dynamic loading of classes, new classes can be instantiated at run-time. Knowledge of these classes is not required at compile-time. This provides flexibility, allowing systems to be extended with new classes at run-time.

Dynamic Loading Facilities provided by the reflective components of the Java runtime environment also allow the loading of classes from files of byte-code. The only restriction placed on this is that the file to be loaded must be a valid class file.

API The rich API provided by the Java development kit contains classes providing functionality for user interfaces (traditionally a facility which has been added to languages after their design), with a comprehensive approach to interaction, using callbacks and listeners. This allows for rapid development of user interfaces, in a simple and reliable fashion.

5.2.2.6 Use of Reflection

Meta-objects are used to represent both the components and the connectors which make up a software system. Communication is managed using operations on communication objects (represented by instantiations of the `Pipe` class).

Pipes have the interface shown in figure 5.2. All of the methods shown are called by the framework in order to control communication between components.

Constructor simply creates an instance of `Pipe`, which connects the two filters given as parameters. The filters communicate with the pipe *via* the ports given.

getInput() and **getOutput()** return the `Filters` which are connected to the `Pipe`.

getInputPort() and **getOutputPort()** return the port numbers that the pipe is connected to on its input and output filters.

equals(Pipe) overrides the `java.lang.Object` method of the same name.

pauseInput() and **pauseOutput()** are used by the framework, when it is reconfiguring a system, to (temporarily) suspend flow either into or out of, the pipe.

inputIsPaused() and **outputIsPaused()** are both used internally by the framework to determine the status of a pipe.

resumeInput() and **resumeOutput()** are the the inverses of the previous pause methods, acting correspondingly.

setOutputDestination() disconnects the pipe from its current output filter, and reconnects it to the specified port on the given filter.

setInputSource() correspondingly allows the setting of the input source to the pipe.

terminate() indicates to the pipe that it is to cease passing data, tidy up, and exit.

hasMessages() returns true if and only if the pipe's message buffer is non-empty.

run() is used to start the pipe's execution.

addPipe() concatenates the given pipe's message queue to the end of the pipe's queue.

5.2.2.7 Implementation of Filters

A `Filter` is a subclass of `java.lang.Thread`, and has the interface shown in figure 5.3. A filter's communication methods are called by the various `Pipes` to which it is connected:

hasMessages() and **isReadyToReceive()** are called by output and input `Pipes`, respectively, to determine whether the filter is ready to communicate.

```
public class Pipe extends Thread
{
   public Pipe(Filter sender, int senderPortNo,
               Filter receiver, int receiverPortNo)
      throws PortInactiveException;

   public synchronized Filter getInput();
   public int getInputPort()

   public synchronized Filter getOutput();
   public int getOutputPort();

   public synchronized void pauseOutput();
   public synchronized void pauseInput();

   public boolean inputIsPaused();
   public boolean outputIsPaused();

   public synchronized void resumeInput();
   public synchronized void resumeOutput();

   public synchronized void setOutputDestination(
                              Filter newOutput, int newOutPort)
                              throws PortInactiveException;

   public synchronized void setInputSource(Filter newInput,
                                           int newInPort)
                              throws PortInactiveException;

   public void terminate();

   public boolean hasMessages();

   public void run();

   public boolean equals(Pipe p);

   public void addPipe (Pipe p);
}
```

Figure 5.2: The Interface of a Pipe.

```
public abstract class Filter extends Thread
{
   public abstract boolean hasMessages(int outPortNo);

   public abstract boolean isReadyToReceive(int inPortNo);

   public abstract Object send(int outPortNo);

   public abstract void receive(int inPortNo,Object message);

   public abstract boolean canActivateInputPort(int portNo);
   public abstract boolean canActivateOutputPort(int portNo);

   public abstract void activateInputPort(int portNo);
   public abstract void activateOutputPort(int portNo);

   public abstract void terminate();
}
```

Figure 5.3: The Interface of a `Filter`.

send() is called by an output pipe in order to receive a message from the filter on a specified port. The filter must return an object corresponding to the message.

receive() is called by an input pipe in order to pass a message into the filter through the given port. The filter handles the object appropriately, probably adding it to an internal buffer for later processing.

canActivateInputPort() and canActivateOutputPort() are called by the framework to determine whether the filter is willing to allow a pipe to be attached to the given ports.

activateInputPort() and **activateOutputPort()** are used by the framework (after verification using the two **can...()** methods) to inform the filter that a pipe has been attached to the corresponding input or output port.

terminate() is called by the framework to indicate that the filter is to cease processing, tidy up, and exit.

Typically, a filter operates as a single thread with a main event-handling loop. The methods which are called by pipes can be thought of as call-back methods, allowing the framework (*via* its pipes) to communicate with the filter. In particular, the methods which cause data to flow into the filter correspond closely with action-type call-back methods as used, for example, in user interface toolkits.

5.2.2.8 User Interface Aspects

The user-interface of the framework (as opposed to the user interface of systems created in the framework) allows the system builder to interact with the architecture of the system. There are two main kinds of action: requests for information, and requests for action.

Requests for information take the form of interrogating pipes or filters in order to determine (and display) properties of these objects. Available information about filters includes the type-hierarchy of the object, and the connection state of its input and output ports. For a pipe, the filters, and ports, to which it is connected can be determined, as can the state of the pipe in terms of number of messages in its internal queue, and whether its input and output is paused or active.

Requests for action are meta-object protocol methods, performing such tasks as inserting filters into pipes, removing pipes, or removing filters.

5.3 Case Studies

This section describes the ways in which the various case studies have motivated the development of the framework. Each case-study has different characteristics, and has motivated different aspects of the framework.

5.3.1 Key Word In Context

As the first, and simplest, case-study, the key-word in context example motivated most of the basic features of the system. Initial features included

simple connectivity between components, using pipes with buffers, providing asynchronous communication.

The key word in context system, as described in section 4.5.1, involves four components, in a simple pipeline. This system, therefore, could be implemented without making use of the concept of ports. This was initially the case.

5.3.2 Markov-Chain

This case study, as documented in section 4.5.2, also involves four components. This also could be implemented using a simple pipe-line style, as opposed to the full pipe-and-filter style described above.

5.3.3 Gas Station

The gas station, described in section 4.5.3, was the most complex example, and required the use of the full pipe-and-filter style. This meant that the framework had to be updated to allow the use of multiple input and output ports for each filter.

5.4 Summary

This chapter has shown the motivation for the creation of a reflective object-oriented framework for managing software, and the requirements which it must fulfil. These requirements will be used in the next chapter to evaluate the framework. The ways in which the three main case studies have been

used to motivate features of the framework have also been described. The framework makes use of the pipe-and-filter model of software. This model and the way in which it is used by the framework has been explained. The way in which the framework makes use of the concept of object-oriented reflection in order to satisfy these requirements has also been described.

Chapter 6

Results and Evaluation

6.1 Introduction

This chapter presents the results of the research. In section 6.2, the results of the three main case studies are presented. In section 6.3, the framework itself is evaluated. Requirements for evaluation are given and then refined into detailed evaluation criteria. These evaluation criteria are then used to evaluate the framework.

6.2 Case Studies

In this section, the three case studies are presented, and the results of the experiments are outlined. Each case study is presented in detail, and the results of evolving each example system are given.

6.2.1 Key Word In Context

The KWIC system was implemented using six classes, as shown in figure 6.1. Four of the classes correspond to components in the architectural design, the remaining two are message classes, used to carry data through the pipes. In the KWIC system, each component class is instantiated exactly once, while message classes are instantiated many times.

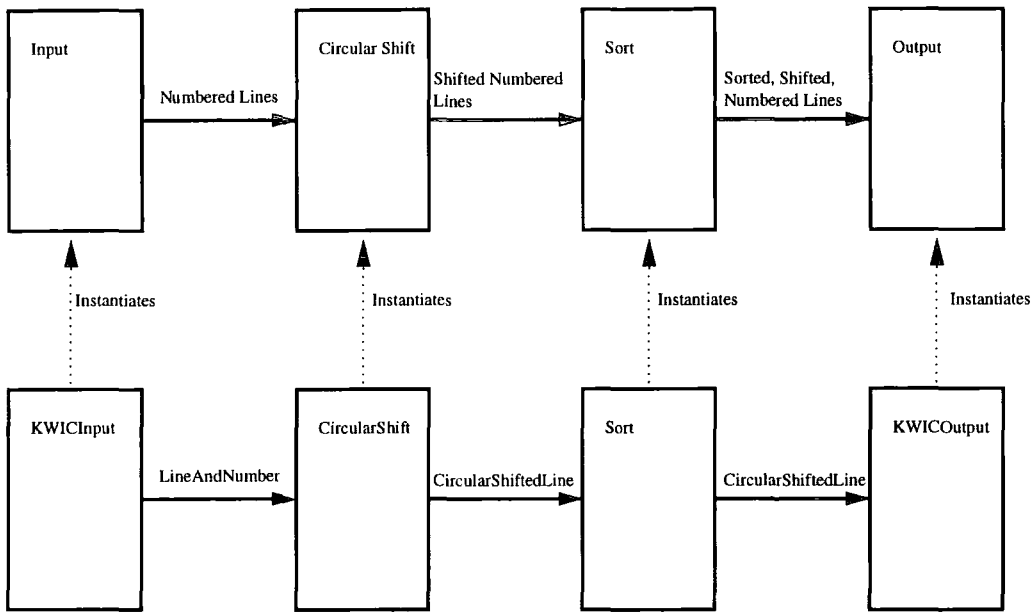

Figure 6.1: Instantiation of Components in the KWIC System

The KWIC implementation showed that the framework was capable of supporting simple interactive software in a steady state (*i.e.*, before evolution has taken place). It also showed that the framework supports run-time visualisation of the architecture of a software system in an appropriate fashion.

Implementing the KWIC system using the framework shows that the framework can be used to construct software, and to support simple evolution. The system can be constructed in either of two ways. It can be built up

using the user interface, starting with an empty system, adding components one at a time, or it can be automated, using the simple Java program shown in figure 6.2.

```
import newarch.*;

public class TestInput
{
   public static void main(String[] args)
                              throws PortInactiveException
   {
       PipeAndFilterSystem sys = new PipeAndFilterSystem();

       InputFile in = new InputFile();
       CircularShift shift = new CircularShift();
       Sorter sort = new Sort();
       KWICOutput output = new KWICOutput();

       sys.addFilter(output, 0, 0);
       sys.addFilter(sort, 0, 0);
       sys.addFilter(shift, 0, 0);
       sys.addFilter(in, 0, 0);
   }
}
```

Figure 6.2: Constructing the KWIC System

In terms of evolution, the KWIC system implementation showed that it is possible to insert components into pipelines, to remove components, and to monitor the addition of pipes into a system, ensuring that pipes are only inserted when the components are compatible. Evolution of the example system is carried out using the graphical interface, which, in turn, calls methods in the framework to make changes to the system.

6.2.2 Markov Chain

The Markov-Chain text-processing system demonstrated the capabilities of the framework in that it can support run-time replacement of modules in a safe manner (for example, replacing the component which manages data storage) while the system is idle. This system also demonstrated that the framework is capable of performing modification at the architectural level.

6.2.3 Gas Station

As the largest and most complex of the case-studies, the gas station example demonstrated (and motivated) most of the features of the framework. It showed that the framework is capable of supporting larger systems with complex (and multiple) inter-component communication, adding and removing components with many connectors attached, and ensuring that a visual representation of the system is maintained at all times.

Figure 6.3 shows the relationships between the objects which implement the gas station and the architectural model.

In common with the previous examples, the example can be either constructed from an empty system using the graphical user interface, or 'scripted' using a simple Java program.

One of the example operations performed on the gas station was the addition of pumps to the system at run-time. This showed that complex operations, involving the addition of many entities (in this case, a component and two pipes), are possible.

6.3 Framework Results

In this section, the framework is evaluated against the criteria from chapter 3. Firstly, the criteria which will be used to evaluate the framework are given (in section 6.3.1), and then, in section 6.3.2, the framework is measured against these criteria.

6.3.1 Evaluation Criteria

The objectives which this research was initially aimed to achieve, as initially presented in section 3.4, are *architectural flexibility*, *higher-level intervention*, *safe changes*, and *architectural visibility*.

Each of these objectives can be refined to evaluation criteria as follows:

Architectural Flexibility The framework must be capable of modelling the architectural level of a software system. This model must describe and show the components and the connectors of a software system, and make them explicit.

Higher-Level Intervention The framework must allow the modification of such a software system (and its corresponding model) at run-time. This modification must be possible at the level of the architectural level.

Safe Changes The framework must ensure that changes to the system are safe (the meaning of 'safe' in this context is explored in section 6.3.1.3).

Architectural Visibility The framework must make the structure of the

system visible, for example by displaying a diagram in a graphical user interface.

The evaluation criteria are thus *architectural modelling* (described in section 6.3.1.1), *high-level modification* (section 6.3.1.2), *safe changes* (section 6.3.1.3, and *structural visibility* (section 6.3.1.4). Each of the objectives is related to at least one (and usually more than one) of these criteria, in the following way. In order for architectural flexibility to be achieved, it is necessary to have both architectural modelling (if the architecture is not modelled, it cannot be controlled) and higher-level modification (the architecture is a higher-level construct). Higher-level intervention is directly dependent on higher-level modification, as well as structural visibility (it must be possible to see the higher-level structure in order to make changes to it). Safe changes, as well as safety itself, require higher-level modification (in order to disallow changes below the level at which the framework. Architectural visibility requires both the modelling of the architecture and structural visibility.

This relationship between the criteria and the objectives is summarised in table 6.1.

Each of these criteria is tackled in sections 6.3.1.1–6.3.1.4. In each section, a high-level set of requirements is given. Each of these requirements is then broken down into a set of concrete evaluation criteria against which the framework can then be easily compared. For ease of comparison, a table summarises the relationships between requirements and evaluation criteria.

Objectives	Evaluation Criteria: Architectural Modelling	High-level Modification	Safe Changes	Structural Visibility
Architectural Flexibility	◉	○		
Higher-level Intervention		○		◎
Safe Changes		●	⊘	
Architectural Visibility	⊖			●

Table 6.1: Relating Objectives to Evaluation Criteria

6.3.1.1 Architectural Modelling

The framework treats a software system as a collection of components and connectors. In order to determine whether the framework supports this, the following requirements must be evaluated:

Accuracy The model of the system which is held by the software must be accurate, corresponding exactly with the actual system which is being modelled.

Updates The model must be updated whenever the system changes; the accuracy must be maintained over evolutions of the system.

Relevance The model of the software system must be relevant: all the important parts of the system must be modelled, and unimportant parts are not to be modelled.

These requirements can be further refined to give the following evaluation criteria. The relationships between the above requirements and the following evaluation criteria are given in table 6.2

Model The set of components which make up the system must be modelled by the framework: every object (component or connector) which appears in the system must be modelled by a corresponding meta-object in the framework's model of the system.

Architecture The architecture of the model must correspond exactly with the architecture of the system being modelled.

Timeliness When a change is made to either the system or the meta-model, the correspondence between the model and the system will be broken. When these events occur, corrections must be made as soon as possible (*n.b.*, no real-time constraints are made here).

Content of model The model of the software must include all the components and connectors which make up the system, and include details of the ports that connectors are attached to.

	Requirements		
Evaluation Criteria	Accuracy	Updates	Relevance
Model	•		•
Architecture	•		
Timeliness	•	•	
Content of Model			•

Table 6.2: Architectural Modelling: Requirements and Evaluation Criteria

6.3.1.2 Higher-Level Modification

Modification can occur at many levels in a software system. Flexibility can be achieved at code-time, compile-time, or run-time. The framework presented here aims at run-time flexibility. The specific goals for flexibility at the

architectural level are allowing insertion and removal of components and connectors.

The requirements for this aim are as follows:

Run-Time Change The framework must allow and support changes at run-time.

Flexibility It must be possible to make meaningful changes to systems using the framework.

These requirements are met by the following evaluation criteria. The relationships between the requirements and the evaluation criteria are summarised in table 6.3.

Run-Time Change The framework must present a user interface at run-time of the system. This interface must allow the system's maintainer to interact with the architecture of the system.

Component Changes It must be possible to make changes to the components in the system (at run-time). It must be possible to remove components from the system, and to insert new components.

Connector Changes It must be possible to change the configuration and number of connectors in the system. It must be possible to remove connectors from the system, and to insert connectors into the system.

	Requirements	
Evaluation Criteria	Run-Time Change	Flexibility
Run-Time Change	◉	
Component Changes		◎
Connector Changes		◎

Table 6.3: Higher-Level Modification: Requirements and Evaluation Criteria

6.3.1.3 Safe Changes

The framework should prevent certain kinds of unsafe changes. It is, obviously, impossible to prevent all kinds of unsafe changes being made to the system, so it is important to be specific about the kinds of changes that are denied. The framework is concerned with architectural modelling and control, so, clearly, it is architectural properties of changes that are under consideration. It is not possible, for example, to prevent that addition of components which function incorrectly (*e.g.*, consider a 'sort' component which sorts into reverse order). Rather, the kinds of changes under consideration are structural.

The requirements, therefore, for safety (as applied in this context), are as follows:

Connectivity The system must operate as one whole individual system.

Connections Components and connectors must be connected properly, in ways for which they are suitable.

The following evaluation criteria encapsulate the above requirements (the relationship between the requirements and evaluation criteria is summarised in table 6.4):

Connectivity The architectural model of the system must be a single connected graph.

Component Connection Each component provides methods which can be called by the framework to determine whether the component is able to accept pipe connections to each port. The framework must prevent any other connections being made.

Pipe Connection Each pipe must be connected to exactly two component; one for each of input to and output from the pipe. These connections must be to a port which handles the relevant type of messages.

	Requirements	
Evaluation Criteria	Connectivity	Connections
Connectivity	•	
Component Connection		•
Pipe Connection		•

Table 6.4: Safe Changes: Requirements and Evaluation Criteria

6.3.1.4 Structural Visibility

In order for the framework to be useful, it must be possible for the maintainer to interact with it. The framework must make visible the architecture of the system which it is modelling, and allow interaction with it. The requirements are:

Visibility The architecture of the system must be presented to the user accurately.

Interaction The user must be able to make changes to the architecture of the system, including adding and removing components and connectors.

These requirements can be refined to the following evaluation criteria (the relationship between the requirements and evaluation criteria is summarised in table 6.5):

Display The framework must display the architecture of the system that is being modelled.

Accuracy The display of the architecture must correspond with the model of the architecture held by the framework, and changes to the model must be reflected by changes to the display.

Interaction The framework must allow the modification of the system, providing means by which the user can add and remove both components and connectors.

	Requirements	
Evaluation Criteria	Visibility	Interaction
Display	•	
Accuracy	•	
Interaction		•

Table 6.5: Structural Visibility: Requirements and Evaluation Criteria

6.3.2 Evaluation

In this section, the framework is evaluated against each of the evaluation criteria given above.

6.3.2.1 Architectural Modelling

The architectural model used by the framework is an array (Java `Vector`) of pipes. The components which make up the system are not directly represented, though methods on the `Pipe` object can be used to interrogate a pipe to determine the components to which it is connected.

The four evaluation criteria are entitled Model, Architecture, Timeliness, and Content of Model. With respect to these criteria, the framework compares as follows:

Model A system is modelled primarily as an array of `Pipe` objects, as these are the primary meta-objects in the framework. By interrogating these objects using the methods `getInput()` and `getOutput` (and also the methods `getInputPort()` and `getOutputPort()`), the framework can determine the entire architecture of the system. In this way, there are two layers to the model; the higher-level is the set of pipes used in the system. This set of pipes is in turn examined to determine the set of components which make up the functionality of the system. Thus, the framework models every entity (component or connector) which is part of the system under consideration. Since every significant feature of a system is modelled, this criterion is satisfied.

Architecture When a new system is created, there are no components and no connectors in the system. Components and connectors can only be added to a system using methods on the framework (such as `addFilter(Pipe p, Filter f, int outPort, int inPort)`) which both update the system and the framework's model of the system. In

this way, the framework's model of the system is automatically updated in the same method as the system itself is updated. The initial (empty) system is correctly modelled by the framework, and every change to the system results in a change to the model, therefore this criterion is satisfied.

Timeliness When a method which updates the system is called, the same method also updates the framework's model of the system. Although no real-time guarantees can be made, the two representations of the system are updated closely enough in order to satisfy this criteria. For example, one method which adds a filter is shown in figure 6.4, where the system and the model are updated in consecutive statements. In every case of update of the system the change to the model occurs within the method which makes the change to the system, therefore this criterion is satisfied for non-real-time systems.

Content of Model As mentioned above, the model of the system includes firstly all the pipes which make up the system, and secondly all the components in the system. These entities, along with the relationships between them, are sufficient to completely model the system. As the empty system is correctly modelled, and each change to the model is exactly equivalent, this criterion is satisfied.

Each of the four criteria in this section (Model, Architecture, Timeliness, and Content of model) has been satisfied to the degree stated above.

6.3.2.2 Higher-Level Modification

The evaluation criteria for higher-level modification are Run-Time Change, Component Changes, and Connector Changes. Comparing the framework to these criteria yields the following:

Run-Time Change The framework's graphical user interface presents a run-time visualisation of the architecture of the system. This allows the user, using menus, to use the meta-object protocol of the framework to make architectural changes to the system at run-time. The user interface calls the methods shown in figure 6.5 in order to perform these operations. As it is possible to make changes at run-time, this criterion is satisfied.

Component Changes Components can be added to and removed from systems using the run-time interface, as described above and shown in figure 6.5. This criterion is satisfied in that components can be added and removed at run-time. However, it is not possible to replace a component and maintain state information held within that component (for example, if a component which counts the messages which pass through it is replaced, the previous total will be lost).

Connector Changes Connectors can be added to and remove from systems in the same way, therefore this criterion is satisfied.

Each of the three criteria in this section (Run-time change, Component change, and Connector change) has been satisfied to the degree stated above.

6.3.2.3 Safe Changes

The evaluation criteria for safety are Connectivity, Component Connection, and Pipe Connection. In this section, the framework is compared to these evaluation criteria.

Connectivity When components are added, they must be connected (*via* a pipe) to a component which is already part of the system. Pipes can only be added between components which are part of the system. So adding entities (components or connectors) is safe. However, if the removal of a pipe breaks the system into two disconnected systems, the framework loses control of one of the systems. The framework does not prevent this kind of unsafe change occurring. This criterion is only partly satisfied.

Component Connection Each component provides a pair of 'safety' methods: `canActivateInputPort(int portNo)`, which is called for input ports, and `canActivateOutputPort(int portNo)` for output ports, which are called by the framework to determine whether the component is capable of being connected in the given manner. This allows the framework to forbid certain kinds of unsafe changes, and so this criterion is satisfied

Pipe Connection When a pipe is added, exactly two components and two ports must be specified, and it is to these components that the new pipe is connected. If a component is removed, any pipes which are would be left disconnected at either end are removed from the system.

Each of the three criteria in this section (Connectivity, Component connection, and Pipe connection) has been satisfied to the degree stated above.

6.3.2.4 Structural Visibility

The evaluation criteria for structural visibility are Display, Accuracy, and Interaction. The framework is measured against these as follows:

Display The framework presents a graphical representation of the system to the user. This criterion is satisfied.

Accuracy The view of the architecture presented to the user is directly derived from the model held by the framework. As shown in section 6.3.2.1, this model is an accurate model of the system's architecture. Hence, the view presented to the user is accurate, and this criterion is satisfied.

Interaction As described in section 6.3.2.2, the user interface allows the user to modify the system which is being modelled by the framework. Therefore this criterion is satisfied.

Each of the three criteria in this section (Display, Accuracy, and Interaction) has been satisfied to the degree stated above.

6.4 Summary

This chapter has presented the results of the case studies, and the framework. From the objectives presented in chapter 3, a detailed set of evaluation criteria have been defined. The framework has been evaluated against this

set of evaluation criteria. This evaluation is summarised in table 6.6. The framework successfully encapsulates a model of software architecture (representing a system by modelling the components and the connectors which constitute it) and allows the maintainer to modify software at the architectural level. Some kinds of unsafe change are forbidden by the framework, though it is possible to make changes which result in a system becoming irreparably fragmented. Because of this, the framework fails to achieve all of the 'safety' requirements. The framework presents a graphical interface which allows the maintainer to make changes to a system interactively and at run-time.

Objective	Evaluation Criteria	Success?
Architectural Modelling	Model	Yes
	Architecture	Yes
	Timeliness	Yes
	Content of Model	Yes
Higher-Level Modification	Run-Time Change	Yes
	Component Changes	Yes
	Connector Changes	Yes
Safe Changes	Connectivity	No
	Component Connection	Yes
	Pipe Connection	Yes
Structural Visibility	Visibility	Yes
	Interaction	Yes

Table 6.6: Evaluation Summary

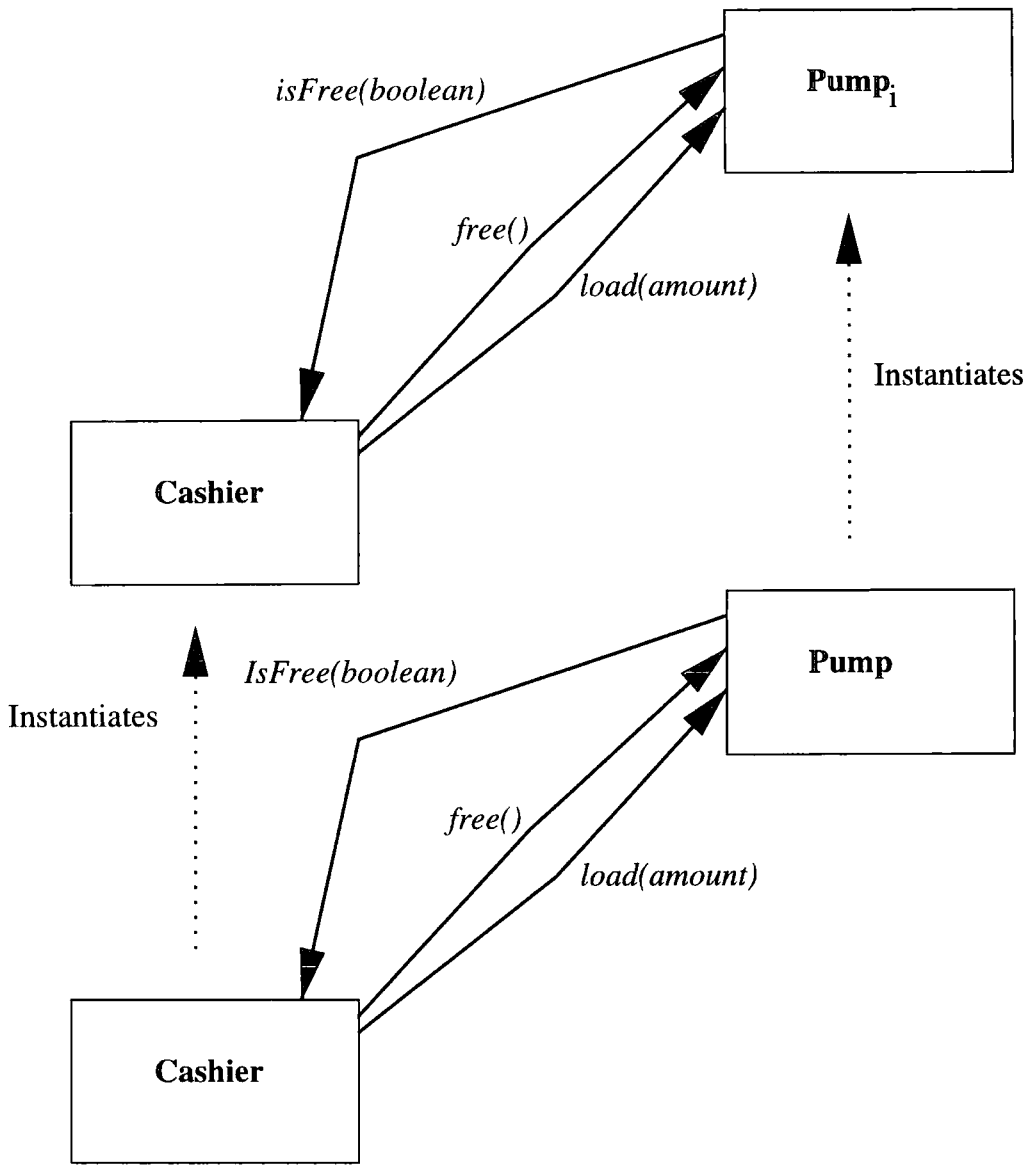

Figure 6.3: Implementation of the Gas Station Example

```
public void addFilter(Pipe p, Filter f,
                      int outPort, int inPort)
   throws PortInactiveException
{
   int thePipeInd;

   synchronized (this.pipes)
   {
      thePipeInd = this.pipes.indexOf(p);
   }

   // pause the pipe
   p.pauseInput();
   p.pauseOutput();

   Filter oldF = p.getOutput();
   int oldPort = p.getOutputPort();

   p.setOutputDestination(f, outPort);

   // Update the system
   Pipe newP = new Pipe(f, inPort, oldF, oldPort);

   // update the model of the system
   this.pipes.insertElementAt(newP, thePipeInd+1);

   newP.start();

   p.resumeInput();
   p.resumeOutput();

   this.updateSystemView();
}
```

Figure 6.4: Adding a filter to a system

```
public void addFilter(Pipe p, Filter f,
                      int outPort, int inPort)
public void removePipe(Pipe p)
public void removeFilter(Filter f)
public void newPipe(Filter inputFilter, int inPort,
                    Filter outputFilter, int outPort)
public void addFilter(Pipe p) throws PortInactiveException
```

Figure 6.5: Methods which update a system

Chapter 7

Conclusion

7.1 Introduction

This chapter summarises the work that has been described in the rest of this thesis. Further, this chapter makes some suggestions for work that could be done in the future to further tackle some of the problems addressed here.

7.2 Problems and Objectives

In the problems which have been addressed were identified in chapter 3: *architectural flexibility*, *higher-level intervention*, *safe changes*, and *architectural visibility*.

7.3 Method

In order to tackle the above problems and achieve the objectives, chapter 4 identified a potential solution, consisting of a reflective framework for managing object-oriented software evolution. The concepts underlying the operation of such a framework were identified, in particular reflection. The three case studies which were used to motivate and evaluate the framework were described. The framework models the architecture of a software system by representing the connectors by which communication takes place, and using a meta-object protocol to interact with the system.

7.4 Process

In chapter 5, the research process was shown. The development of the framework and the associated case studies was described, and the model of architecture used in the framework was shown. The implementation of the framework was shown in detail, and the two main categories of entity (pipe and filter) described. The framework has the following criteria for success:

- The framework must allow the entities which make up a software system, and the means by which these entities interact, to be modified at run-time.
- The framework must allow intervention at a higher level of abstraction than source code statements.
- The framework must ensure that safety properties are maintained.

- The structure of a software system implemented using the framework must be made visible at run-time.

These criteria were later used to determine the way in which the framework was evaluated.

The development of the case studies and the framework gave a greater understanding of the problems and issues involved in the satisfaction of the objectives identified in chapter 3. For example, the development of the gas station framework provided insights into managing the modelling of systems with multiple interconnections between components.

7.5 Results and Evaluation

In chapter 6, the results of the research were presented. The case study results were presented first. A detailed set of objectives were developed, and the framework was examined with respect to these, which can be summarised as *architectural flexibility*, *higher-level intervention*, *safe changes*, and *architectural visibility*.

Each of these objectives was broken down into a set of requirements and then into concrete evaluation criteria, summarised as follows;

Architectural Modelling Model, architecture, timeliness, and content of model.

Higher-Level Modification Run-time change, component changes, and connector changes.

Safe Changes Connectivity, component connection, and pipe connection.

Structural Visibility Display, accuracy, and interation.

The framework was compared against these evaluation criteria, and the results are summarised in figure 7.1. This table shows that the framework satisfies the evaluation criteria, apart from the connectivity evaluation criteria in the "safe changes" section. This criterion requires greater protection against the breaking of a system into two or more disconnected parts when a component or a connector is removed. In the cases of the other evaluation criteria, the framework behaves satisfactorily. For example, the framework contains a model of the architecture which is complete enough to allow a maintainer to interact with the system at the architectural level.

Objective	Success?
Architectural Modelling	Yes
Higher-Level Modification	Yes
Safe Changes	Partly
Structural Visibility	Yes

Table 7.1: Evaluation Summary

The problems which were identified in chapter 3 have thus been addressed satisfactorily, with the exception that it is still possible to make some kinds of unsafe changes to a system which is implemented using the framework.

7.6 Further Work

There are two categories of further work; work which can be done in order to further satisfy the objectives given here, and extensions in new directions.

7.6.1 Greater Satisfaction

As shown in chapter 6, the framework does not entirely fulfil the objectives given in chapter 3, in particular with respect to safety.

In order to produce a framework which better satisfies the objectives given above, the following work would be useful:

- A greater understanding of the concepts of safety, and how further categories of safety can be ensured, in order to determine whether the framework is addressing the problems which are important in system operation. Further work in this area would address the areas of system integrity, and how a system should behave when components are removed.

- An extended meta-object protocol, allowing greater examination of a system by a user of the framework. This work would involve increasing the amount of information recorded by the system, for example the history of connections between components.

- Allowing the framework to handle a greater range of architectural styles.

- Handling replacement of components; allowing the upgrading of a component during its operation without losing its data content.

The framework does not satisfy all the evaluation criteria for safety. This became apparent when carrying out the gas station case study. The lack of proper safety checking can lead to unsafe modifications being carried out,

and thus lead to undesired behaviour of a system. In order to improve the framework to satisfy this goal, extra checks could be added to the methods which are used to add and remove components and connectors, to determine whether an unsafe operation has been attempted. Further, a separate safety-monitoring entity could also be created, which would constantly monitor the state of the system and prevent unsafe changes being made.

The meta-object protocol could be extended in many ways. Perhaps most useful would be including operations to handle the transfer of internal state between instantiations of components when a component is replaced. This would allow components to be upgraded without causing a loss of state. In order to tackle this, it would be necessary to introduce a common mechanism for representing the internal state of a component in a portable manner, to allow a new component to be 'primed' with the state of an earlier version. Particular attention would have to be paid to cases where the internal representation of state differs between versions of a component.

The framework currently handles systems which are exclusively implemented in the pipe-and-filter style. There are many other styles, and many systems use more than one style. To allow this, the framework would have to be extended to include different types of connectors and components. Further safety measures would be required to prevent incompatibilities (for example, a pipe could not be connected to a component which only allows call-and-return connectors). This could be achieved by segmenting systems into parts, each of which is implemented in one particular style. Each of these parts could then be composed into one final system.

7.6.2 New Directions

Work which can lead the framework to satisfy further objectives could involve:

- Allowing scripting, *i.e.*, automatable construction of systems from high-level descriptions. For example, these descriptions could be given in Darwin.
- The use of patterns, as high-level templates for systems could be introduced.
- Monitoring the evolution of a system, in order to allow undoing of changes, and 'snapshots' of the state of a system.

7.7 Summary

This chapter has summarised the work conducted. The content of the thesis was described, and then a set of ideas for further work presented. The main result of this work is a reflective object-oriented framework for enabling and managing run-time software evolution.

Appendix A

Implementation Details

A.1 Pipes

This section shows the Java interface to the Pipe class, described in section 4.4.1.

```
public class Pipe extends Thread
{
   /** Constructor: by default, messages are not echoed.
     *  @param sender  The object which writes to the pipe
     *  @param receiver The object which receives data
     *                                    from the pipe
     */
   public Pipe(Filter sender, int senderPortNo,
               Filter receiver, int receiverPortNo)
      throws PortInactiveException
```

```
/** Constructor which allows specification of verbosity
 * (whether messages are echoed to the screen)
 * @param sender  The object which writes to the pipe
 * @param receiver The object which receives data
 *                                    from the pipe
 * @param beVerbose if true, echo all messages
 *                                    to standard output
 */
public Pipe(Filter sender, int senderPortNo,
            Filter receiver, int receiverPortNo,
            boolean beVerbose)
    throws PortInactiveException

/** Determine which <tt>Filter</tt> is connected to the
  * input of the pipe.
  */
public synchronized Filter getInput()


/** Determine which port input is coming from/
  */
public int getInputPort()
```

```
/** Determine which <tt>Filter</tt> is connected to the
  * output of the pipe.
  */
public synchronized Filter getOutput()


/** Determine which port the output is going to.
  */
public int getOutputPort()

/** Pause communication from the pipe.
  */
public synchronized void pauseOutput()

/** Pause communication to the pipe
  * (i.e., accept no more input).
  */
public synchronized void pauseInput()

/** Determine whether the pipe's input is paused.
  */
public boolean inputIsPaused()

/** Determine whether the pipe's output is paused.
  */
```

```
public boolean outputIsPaused()

/** Resume communication to the pipe (i.e., accept input).
  */
public synchronized void resumeInput()

/** Resume communication from the pipe (i.e., resume sending
  * things.
  */
public synchronized void resumeOutput()

/** Change the destination for output from the pipe.
  */
public synchronized void setOutputDestination(
                                    Filter newOutput,
                                    int newOutPort)
                            throws PortInactiveException

/** Set the source for input to the pipe
  */
public synchronized void setInputSource(Filter newInput,
                                        int newInPort)
   throws PortInactiveException

/** kill the pipe
```

```
     */
    public void terminate()

    public boolean hasMessages()



    /** Start the pipe going.
      * Calls <tt>run</tt> for the input and
      * output components (if they are
      * not already executing).
      */
    public void run()

    /** Add a pipe in front of this pipe.
      */
    public void addPipe (Pipe p)
}
```

A.2 Filter

This section shows the interface to the `Filter` class, as described in section 4.4.1.1.

```
package newarch;

/** Filter: an abstract class of objects that can be
```

```
  * connected via a <tt>Pipe</tt>.
  * The programmer should sub-class <tt>Filter</tt> and provide
  * the methods from this class and the <tt>run</tt> method
  * from the <tt>Thread<tt> class.
  * @see java.lang.Thread
  * @see Pipe
  * @author Stephen Rank
  */

public abstract class Filter extends Thread
{
   /** Called by the pipe to query whether the object is ready
     * to send a message.
     */
   public abstract boolean hasMessages(int outPortNo);

   /** Called by the pipe to obtain a message
     */
   public abstract Object send(int outPortNo);

   /** Called by the pipe to query whether the object is ready
     * to receive a message.
     */
   public abstract boolean isReadyToReceive(int inPortNo);
```

```
    /** Called by the pipe with the message object
      * as a parameter.
      * @param message: the Object to send as a message
      */
    public abstract void receive(int inPortNo,Object message);

     /** Called by a Pipe before activating ports.  Returns
       *  true iff the given port number can be activated.
       */
     public abstract boolean canActivateInputPort(int portNo);
     public abstract boolean canActivateOutputPort(int portNo);

     /** Called by a pipe to perform the activation
      */
     public abstract void activateInputPort(int portNo);
     public abstract void activateOutputPort(int portNo);

    /** This method should clean up and end the thread
      */
    public abstract void terminate();
}
```

Bibliography

[Abelson et al., 1985] Abelson, H., Sussman, G. J., and Sussman, J. (1985). *Structure and Interpretation of Computer Programs.* Electrical Engineering and Computer Science Series. M.I.T. Press.

[Alexander et al., 1977] Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Angel, S. (1977). *A Pattern Language: Towns–Buildings–Construction.* Oxford University Press.

[Amador et al., 1991] Amador, J., de Vicente, B., and Alonso, A. (1991). Dynamically replaceable software: A design method. In van Lamsweerde, A. and Fuggetta, A., editors, *Proceedings of the 3rd European Software Engineering Conference, E.S.E.C.*, volume 550 of *Lecture Notes in Computer Science*, pages 210–228, Milan, Italy. Springer-Verlag.

[Banker and Slaughter, 1997] Banker, R. D. and Slaughter, S. A. (1997). A field study of scale economies in software maintenance. *Management Science*, 43(12):1709–1725. December.

[Bass et al., 1998] Bass, L., Clements, P., and Kazman, R. (1998). *Software Architecture in Practice.* S.E.I. Series in Software Engineering. Addison-Wesley.

[Baxter and Pidgeon, 1997] Baxter, I. and Pidgeon, C. W. (1997). Software change through design maintenance. In *Proceedings of the 1997 International Conference on Software Maintenance (ICSM '97)*, pages 250–259. I.E.E.E.

[Beck and Johnson, 1994] Beck, K. and Johnson, R. (1994). Patterns generate architectures. In *Proceedings of ECOOP 1994*, Lecture Notes in Computer Science, pages 139–149. Springer-Verlag.

[Bennett and Rajlich, 2000] Bennett, K. H. and Rajlich, V. T. (2000). A staged model for the software life cycle. *IEEE Computer*, 33(7):66–71.

[Bihari and Schwan, 1991] Bihari, T. E. and Schwan, K. (1991). Dynamic adaptation of real-time software. *A.C.M. Transactions on Computer Systems*, 9(2):143–174.

[Boehm, 1988] Boehm, B. (1988). A spiral model for software development and enhancement. *Computer*, 21(5):61–72.

[Bosch, 1999] Bosch, J. (1999). Evolution and composition of reusable assets in product-line architectures: A case study. In *Proceedings of the First Working IFIP Conference on Software Architecture*, pages 321–340.

[Boyapati, 2002] Boyapati, C. (2002). Towards an extensible virtual machine. Technical Report MIT-LCS-TR-842, MIT Laboratory for Computer Science.

[Brandt, 1995] Brandt, S. (1995). Reflection in a statically typed and object oriented language – A meta-level interface for BETA. Technical report, Computer Science Department, Aarhus University, Denmark. `http://www.daimi.au.dk/~beta/Papers/sbrandt/betamli.html`.

[Brandt and Schmidt, 1995] Brandt, S. and Schmidt, R. W. (1995). The design of a meta-level architecture for the BETA language. In *Proceedings of the ECOOP Workshop on Advances in Metaobject Protocols and Reflection (META'95)*.

[Brooks, 1995] Brooks, F. P. (1995). *The Mythical Man-Month.* Addison-Wesley.

[Burd and Munro, 1998] Burd, E. and Munro, M. (1998). Assisting human understanding to aid the targeting of necessary reengineering work. In *Proceedings of the Fifth Working Conference on Reverse Engineering (WCRE'98)*, pages 2–9, Honolulu, Hawaii. IEEE Computer Society.

[Buschmann, 1996] Buschmann, F. (1996). Reflection. In Vlissides, J. M., Coplien, J. O., and Kerth, N. L., editors, *Pattern Languages of Program Design*, pages 271–294. Addison Wesley.

[Cazzola et al., 1998] Cazzola, W., Savigni, A., Sosio, A., and Tisato, F. (1998). Architectural reflection: Bridging the gap between a running system and its specification. In *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*, Florence, Italy.

[Danvy and Malmkjær, 1988] Danvy, O. and Malmkjær, K. (1988). Intensions and extensions in a reflective tower. In *Proceedings of LFP88, the*

1988 ACM Conference on Lisp and Functional Programming, pages 327–341.

[Dijkstra, 1968] Dijkstra, E. W. (1968). The structure of the "T.H.E."-multiprogramming system. *Communications of the A.C.M.*, 11(5):341–346. Reprinted in Communications of the A.C.M. 26(1), January 1983.

[Ducasse and Günter, 1998] Ducasse, S. and Günter, M. (1998). Coordination of active objects by means of explicit connectors. In *Proceedings of the DEXA'98 Workshop.* Available on the World-Wide Web at `http://www.iam.unibe.ch/~ducasse/PubHTML/newpage.html#repository`.

[Fayad and Schmidt, 1997] Fayad, M. E. and Schmidt, D. C. (1997). Object-oriented application frameworks. *Communications of the A.C.M.*, 40(10):32–38.

[Fyson and Boldyreff, 1998] Fyson, M. J. and Boldyreff, C. (1998). Using application understanding to support impact analysis. *Journal of Software Maintenance: Research and Practice*, 10(2):93–110.

[Gacek et al., 1995] Gacek, C., Abd-Allah, A., Clark, B. K., and Boehm, B. (1995). On the definition of software system architecture. In Garlan, D., editor, *Proceedings of the First International Workshop on Architectures for Software Systems*, pages 85–95, Seattle, Washington.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

[Garlan, 1998] Garlan, D. (1998). Higher-order connectors. Presented at the Workshop on Compositional Software Architectures, Monterey, California. Available on the World-Wide Web at `http://www.cs.cmu.edu/afs/cs/project/able/ftp/hoc-omg98/hoc-omg98.pdf`.

[Garlan et al., 1995] Garlan, D., Allen, R., and Ockerbloom, J. (1995). Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings of the Seventeenth Interntional Conference on Software Engineering*, pages 179–185, Seattle, Washington, U.S.A.

[Garlan and Shaw, 1994] Garlan, D. and Shaw, M. (1994). An introduction to software architecture. Technical Report CMU/SEI-94-TR-21 or

ESC-TR-94-24, Software Engineering Institute, Carnegie Mellon University. Also published in *Advances in Software Engineering* Volume 1, ed. V. Ambrolia and G. Tortora, 1993.

[Goldberg and Robson, 1983] Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and its Implementation.* Addison-Wesley.

[Golm and Kleinöder, 1998] Golm, M. and Kleinöder, J. (1998). metaXa and the future of reflection. In *Proceedings of the Workshop on Reflective Programming in C++ and Java.* Available on the World-Wide Web at `http://www4.informatik.uni-erlangen.de/TR/pdf/TR-I4-98-09.pdf`.

[Heineman, 1998] Heineman, G. T. (1998). Adaptation and software architecture. In *Proceedings of the 3rd Annual International Workshop on Software Architecture (ISAW-3)*, pages 61–64, Orlando, Florida.

[I.E.E.E., 1994] I.E.E.E. (1994). *I.E.E.E. Software Engineering Standards Collection.* I.E.E.E. Press.

[Jackson, 1998] Jackson, M. (1998). Will there ever be software engineering? *IEEE Software*, 15(1):36–39.

[Johnson, 1997] Johnson, R. E. (1997). Frameworks = (Components + patterns). *Communications of the A.C.M.*, 40(10):39–42.

[Kernighan and Pike, 1999] Kernighan, B. W. and Pike, R. (1999). *The Practice of Programming.* Addison Wesley Longman.

[Kiczales, 1996] Kiczales, G. (1996). Beyond the black box: Open implementation. *IEEE Software*, 13(1):8–11.

[Kiczales et al., 1993] Kiczales, G., Ashley, J. M., Rodriguez, L., Vahdat, A., and Bobrow, D. G. (1993). Metaobject protocols: Why we want them and what else they can do. In Paepcke, A., editor, *Object-Oriented Programming: The CLOS Perspective*, pages 101–118. M.I.T. Press.

[Kiczales et al., 1991] Kiczales, G., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol.* M.I.T. Press.

[Kirby et al., 1998] Kirby, G., Morrison, R., and Stemple, D. (1998). Linguistic reflection in Java. *Software—Practice and Experience*, 28(10):1045–1077.

[Kramer and Magee, 1985] Kramer, J. and Magee, J. (1985). Dynamic configuration for distributed systems. *I.E.E.E. Transactions on Software Engineering*, SE-11(4):424–436.

[Lehman, 1979] Lehman, M. M. (1979). On understanding law, evolution and conservation in the large program, life cycle. *Journal of Systems and Software*, 1:213–221.

[Lehman, 1989] Lehman, M. M. (1989). Uncertainty in computer application and its control through the engineering of software. *Journal of Software Maintenance: Research and Practice*, 1(1):3–27.

[Lehman, 1996] Lehman, M. M. (1996). Laws of software evolution revisited. In *Proceedings of EWSPT96*, number 1149 in Lecture Notes in Computer Science, pages 108–124. Springer-Verlag.

[Lehman, 1997] Lehman, M. M. (1997). Feedback in the software process. Position Paper at the SEA Workshop: *Research Directions in Software Engineering*, Imperial College, London.

[Lehman, 1998a] Lehman, M. M. (1998a). FEAST, FEAST/1 and FEAST/2. Feedback, Evolution And Software Technology, Magnet Seminar, Tel Aviv, Israel. Also presented at the University of Durham, January 1999.

[Lehman, 1998b] Lehman, M. M. (1998b). Software's future: Managing evolution. *IEEE Software*, 15(1):40–44.

[Lehman and Belady, 1985a] Lehman, M. M. and Belady, L. A. (1985a). *Program Evolution: Processes of Software Change.* Number 27 in APIC Studies in Data Processing. Academic Press.

[Lehman and Belady, 1985b] Lehman, M. M. and Belady, L. A. (1985b). Programs, life cycles and laws of software evolution. In *Program Evolution: Processes of Software Change*, number 27 in APIC Studies in Data Processing, pages 393–449.

[Lehman et al., 1997] Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., and Turski, W. M. (1997). Metrics and laws of software evolution—The nineties view. In Eman, K. E. and Madhavji, N. H., editors, *Elements of Software Process Assessment and Improvement*, pages 20–32, Albuquerque, New Mexico. IEEE CS Press.

[Lientz and Swanson, 1980] Lientz, B. P. and Swanson, E. B. (1980). *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations.* Addison-Wesley.

[Luckham et al., 1995] Luckham, D. C., Kenney, J. J., Augustin, L., Vera, J., Bryan, D., and Mann, W. (1995). Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–359.

[Lung et al., 1997] Lung, C.-H., Bot, S., Kalaichelvan, K., and Kazman, R. (1997). An approach to software architecture analysis for evolution and reusability. In *Proceedings of CASCON '97*, Toronto, Ontario, Canada.

[Maeda et al., 1997] Maeda, C., Lee, A., Murphy, G., and Kiczales, G. (1997). Open implementation analysis and design. In *Proceedings of the 1997 Symposium on Software Reusability*, pages 44–52, Boston, Massachusetts, United States. ACM Press.

[Magee et al., 1995] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. (1995). Specifying distributed software architectures. In *Proceedings of 5th European Software Engineering Conference (ESEC 95)*, Sitges, Spain.

[Mätzel and Bischofberger, 1996] Mätzel, K.-U. and Bischofberger, W. (1996). Evolution of object systems – How to tackle the slippage problem. In Mätzel, K.-U. and Frei, H. P., editors, *Computer Science Research at Ubilab, Research Projects 1995/96; Proceedings of the Ubilab Conference '96*, pages 99–119, Universitätsverlag Konstanz.

[McConnell, 1993] McConnell, S. (1993). *Code Complete: A Practical Handbook of Software Construction.* Microsoft Press.

[McDermid, 1991] McDermid, J. A., editor (1991). *Software Engineers Reference Book.* Butterworth-Heinemann.

[Mendhekar and Friedman, 1993] Mendhekar, A. and Friedman, D. P. (1993). Towards a theory of reflective programming languages. In *Proceedings of the 1993 OOPSLA Workshop on Reflection and Meta-level Architectures.*

[Oreizy, 1998] Oreizy, P. (1998). Issues in modeling and analyzing dynamic software architectures. In *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis*, Marsala, Sicily, Italy.

[Oreizy and Medvidovic, 1998] Oreizy, P. and Medvidovic, N. (1998). Architecture-based runtime software evolution. In *Proceedings of the International Conference on Software Engineering 1998 (ICSE'98)*, Kyoto, Japan.

[Oreizy et al., 1998a] Oreizy, P., Medvidovic, N., and Taylor, R. N. (1998a). Software architecture and component technologies: Bridging the gap. In *Proceedings of the OMG-DARPA Workshop on Compositional Software Architectures*, Monterey, California.

[Oreizy et al., 1998b] Oreizy, P., Rosenblum, D. S., and Taylor, R. N. (1998b). On the role of connectors in modelling and implementing software architectures. Technical Report UCI-ICS-98-04, Department of Information and Computer Science, University of California, Irvine, California.

[Oreizy and Taylor, 1998a] Oreizy, P. and Taylor, R. N. (1998a). On the role of software architectures in runtime system reconfiguration. In *Proceedings of the International Conference on Configurable Distributed Systems (ICCDS 4)*, Annapolis, Maryland.

[Oreizy and Taylor, 1998b] Oreizy, P. and Taylor, R. N. (1998b). On the role of software architectures in runtime system reconfiguration. *I.E.E. Proceedings-Software*, 145(5):137–145.

[Parnas, 1972] Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the A.C.M.*, 15(12):1053–1058.

[Perry and Wolf, 1992] Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52.

[Pigoski, 1996] Pigoski, T. M. (1996). *Practical Software Maintenance.* John Wiley and Sons.

[Prieto-Diaz and Neighbours, 1986] Prieto-Diaz, R. and Neighbours, J. M. (1986). Module interconnection languages. *Journal of Systems and Software*, 6:307–334.

[Ribeiro-Justo and Cunha, 1999] Ribeiro-Justo, G. R. and Cunha, P. R. F. (1999). An architectural application framework for evolving distributed systems. *Journal of Systems Architecture*, 45(15):1375–1384.

[Rubini, 1997] Rubini, A. (1997). The sysctl interface. *Linux Journal*, 41. Available on the World-Wide Web at `http://www2.linuxjournal.com/lj-issues/issue41/2365.html`.

[Sabry, 1998] Sabry, A. (1998). A programming language perspective to compositional software architectures. In *Workshop on Compositional Software Architectures.* Available at `http://www.objs.com/workshops/ws9801/papers/paper042.html`.

[Schneidewind et al., 1999] Schneidewind, N., Kitchenham, B., Niessick, F., Singer, J., von Mayrhauser, A., and Yang, H. (1999). Resolved: "Software maintenance is nothing more than another form of development". In *Proceedings of the International Conference on Software Maintenance 1999*, pages 63–64, Oxford, U.K. I.E.E.E., I.E.E.E. Computer Society Press.

[Schneidewind, 1987] Schneidewind, N. F. (1987). The state of software maintenance. *I.E.E.E. Transactions on Software Engineering*, SE-13(3):303–310.

[Segal and Frieder, 1989] Segal, M. E. and Frieder, O. (1989). Dynamic program updating: A software maintenance technique for minimizing software downtime. *Journal of Software Maintenance: Research and Practice*, 1(1):59–79.

[Shaw, 1993] Shaw, M. (1993). Procedure calls are the assembly language of software interconnection: Connectors deserve first class status. Technical Report CMU/SEI-94-TR-2, Software Engineering Institute, Carnegie Mellon University. Presented at the Workshop of Software Design, 1994. Published in the proceedings: LNCS 1994.

[Shaw, 1995] Shaw, M. (1995). Architectural issues in software reuse: It's not just the functionality, it's the packaging. In *Proceedings of the I.E.E.E. Symposium on Software Reusability.*

[Shaw et al., 1995] Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., and Zelesnik, G. (1995). Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335.

[Shaw and Garlan, 1994] Shaw, M. and Garlan, D. (1994). Characteristics of higher-level languages for software architectures. Technical Report CMU-CS-94-210, Department of Computer Science, Carnegie Mellon University.

[Shaw and Garlan, 1996] Shaw, M. and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall.

[Smith, 1982] Smith, B. C. (1982). *Reflection and Semantics in a Procedural Language.* PhD thesis, M.I.T. Laboratory for Computer Science. M.I.T. Technical Report MIT/LCS/TR-272.

[Smith, 1999] Smith, D. D. (1999). *Designing Maintainable Software.* Springer-Verlag.

[Sobel and Friedman, 1996] Sobel, J. M. and Friedman, D. P. (1996). An introduction to reflection-oriented programming. In *Proceedings of Reflection '96*, San Francisco.

[Steindl, 1997] Steindl, C. (1997). Reflection in Oberon. In Mössenböck, H., editor, *Modular Programming Languages: Joint Modular Programming Languages Conference, JMLC'97*, number 1204 in Lecture Notes in Computer Science, pages 282–296, Linz, Austria. Springer-Verlag.

[Szyperski, 1997] Szyperski, C. (1997). *Component Software: Beyond Object-Oriented Programming.* A.C.M. Press.

[Takang and Grub, 1996] Takang, A. A. and Grub, P. A. (1996). *Software Maintenance: Concepts and Practice.* International Thomson Computer Press.

[von Mayrhauser and Vans, 1995] von Mayrhauser, A. and Vans, A. (1995). Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55.

[Welch and Stroud, 1998] Welch, I. and Stroud, R. (1998). Adaptation of connectors in software architectures. In *ECOOP'98 Workshop on Component-Oriented Programming*, Brussels, Belgium.

[Welch and Stroud, 2000] Welch, I. and Stroud, R. (2000). Using reflection as a mechanism for enforcing security policies in mobile code. In *Proceedings of ESORICS.*

[Welch and Stroud, 2001] Welch, I. and Stroud, R. (2001). Kava – Using bytecode rewriting to add behavioural reflection to java. In *Proceedings of USENIX Conference on Object-Oriented Technology.*